# GPlace3.0: Routability-Driven Analytic Placer for UltraScale FPGA Architectures

ZIAD ABUOWAIMER, DANI MAAROUF, TIMOTHY MARTIN, JEREMY FOXCROFT, GARY GRÉWAL, SHAWKI AREIBI, and ANTHONY VANNELLI, University of Guelph, Canada



Optimizing for routability during FPGA placement is becoming increasingly important, as failure to spread and resolve congestion hotspots throughout the chip, especially in the case of large designs, may result in placements that either cannot be routed or that require the router to work excessively hard to obtain success. In this article, we introduce a new, analytic routability-aware placement algorithm for Xilinx UltraScale FPGA architectures. The proposed algorithm, called GPlace3.0, seeks to optimize both wirelength and routability. Our work contains several unique features including a novel window-based procedure for satisfying legality constraints in lieu of packing, an accurate congestion estimation method based on modifications to the pathfinder global router, and a novel detailed placement algorithm that optimizes both wirelength and external pin count. Experimental results show that compared to the top three winners at the recent ISPD'16 FPGA placement contest, GPlace3.0 is able to achieve (on average) a 7.53%, 15.15%, and 33.50% reduction in routed wirelength, respectively, while requiring less overall runtime. As well, an additional 360 benchmarks were provided directly from Xilinx Inc. These benchmarks were used to compare GPlace3.0 to the most recently improved versions of the first- and second-place contest winners. Subsequent experimental results show that GPlace3.0 is able to outperform the improved placers in a variety of areas including number of best solutions found, fewest number of benchmarks that cannot be routed, runtime required to perform placement, and runtime required to perform routing.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Physical design (EDA)**; **Placement**; *Wire routing*; Partitioning and floorplanning;

Additional Key Words and Phrases: Placement, field programmable gate array, congestion, routing-aware, heterogeneous, ultrascale architecture

**ACM Reference format:**
Ziad Abuowaimer, Dani Maarouf, Timothy Martin, Jeremy Foxcroft, Gary Gréwal, Shawki Areibi, and Anthony Vannelli. 2018. GPlace3.0: Routability-Driven Analytic Placer for UltraScale FPGA Architectures. *ACM Trans. Des. Autom. Electron. Syst.* 23, 5, Article 66 (October 2018), 33 pages.
https://doi.org/10.1145/3233244

## 1 INTRODUCTION

*Field Programmable Gate Arrays (FPGAs)* continue to find increasingly wide use in commercial products, due to their flexibility and versatility. However, within the FPGA *Computer-Aided Design (CAD)* flow, *placement* remains one of the most important, time-consuming steps. Given a

Authors' addresses: Z. Abuowaimer (corresponding author), D. Maarouf, T. Martin, J. Foxcroft, G. Gréwal, S. Areibi, and A. Vannelli, University of Guelph, 50 Stone Rd. E, Guelph, ON, N1G 2W1, Canada; emails: {abuowaiz, dmaarouf, tmarti14, jfoxcrof, ggrewal, sareibi, vannelli}@uoguelph.ca.

1084-4309/2018/10-ART66 $15.00
https://doi.org/10.1145/3233244

netlist representation of a circuit, placement involves mapping functional blocks in the netlist to legal locations on the FPGA in a way that optimizes one (or more) objectives. The most basic objective involves minimizing the sum of the total (estimated) *wirelength* for each net. However, as commodity FPGAs grow in size and complexity, *routability* is also becoming an increasingly important placement objective due to the fixed interconnect available in different regions of the FPGA. Placements that fail to resolve congestion throughout the FPGA often incur subsequent routing demands that exceed the prefabricated routing resources available in certain regions of the device. The end result is that the ensuing routing step either fails, or the router is forced to work unreasonably hard to obtain success. Either scenario is undesirable, as placement and routing runtimes are already excessive [2].

We have developed a routing-aware analytic placer for Xilinx UltraScale FPGAs that consistently generates high-quality placements while exhibiting good runtime performance. We have targeted UltraScale FPGAs because they are large, state-of-the-art devices containing many heterogeneous features that make finding even a legal placement challenging. Unlike traditional approaches [1, 20, 21], our approach does not emphasize *packing* early in the placement process. This is because we have found that when circuits contain large numbers of flip-flops and control sets, performing initial (tight) packing can handcuff later optimization steps. Instead, we start with a flat placement and employ a unique window-based algorithm to legalize the placement throughout the placement flow. The final components of any slice are not determined until the end of the flow. By avoiding packing early in the placement process, we aim to provide later optimizations more flexibility and opportunities to improve wirelength and routability. Although not the best strategy for every circuit, this approach performs very well across the large number of benchmarks used in this article.

Our work also contains several other unique features. In contrast to approaches that rely on *Application Specific Integrated Circuit (ASIC)* global routers for estimating congestion, like References [13] and [12], we employ a global router based on modifications made to the FPGA global router, PathFinder [15]. The primary focus here is on improving accuracy when predicting local routing resource demand. We have also developed a detailed placement algorithm, based on enhancements to *Independent-Set Matching (ISM)* [12]. The aim of traditional ISM is on minimizing wirelength following global placement. Our enhancements enable ISM to minimize wirelength and external pin count, with the goal of improving both wirelength and routability.

### 1.1 Contribution

The proposed placement algorithm, called GPlace3.0, seeks to optimize both wirelength and routability. The main contributions of our work are listed below:

- A novel window-based bi-partitioning legalization procedure for legalizing flat placements is proposed. The procedure satisfies all hard constraints imposed by the UltraScale architecture, while minimizing cell displacement. Within GPlace3.0, it eliminates the need for an initial packing step and allows the placer to optimize globally.
- A faster, more accurate version of the pathfinder global router [15] is proposed (mPFGR) for performing congestion estimation. Within GPlace3.0, the mPFGR congestion estimation is used to guide cell inflation/deflation optimizations.
- A dual objective ISM algorithm is proposed for improving both wirelength and external pin count. Within GPlace3.0 it is used to perform detailed placement.
- Experimental results based on the ISPD'16 Routability-Driven FPGA Placement Contest benchmark suite show that GPlace3.0 outperforms the top three [23] contest winners with respect to runtime, routed wirelength, and number of routable placements.

—Experimental results based on 360 additional benchmarks received from Xilinx Inc. show that GPlace3.0 also outperforms the latest (improved) placers from the top three contest winners [12, 13, 17, 18] across a wide range of metrics, including routed wirelength, number of best solutions found, fewest number of unroutable placements, and placement and routing runtimes.

Although the placement techniques described in this article are evaluated using benchmarks aimed at a Xilinx UltraScale device, the proposed techniques are general enough to be easily adapted to other architectures, or to FPGAs from other vendors. The remainder of the article is organized as follows: Section 2 gives a brief review of the literature. Section 3 explains the background and the key differences between the three ISPD '16 contest winners. Section 4 highlights the overall structure and optimizations in GPlace3.0. Results obtained with this approach are explained in Section 5. Section 6 contains our conclusions.

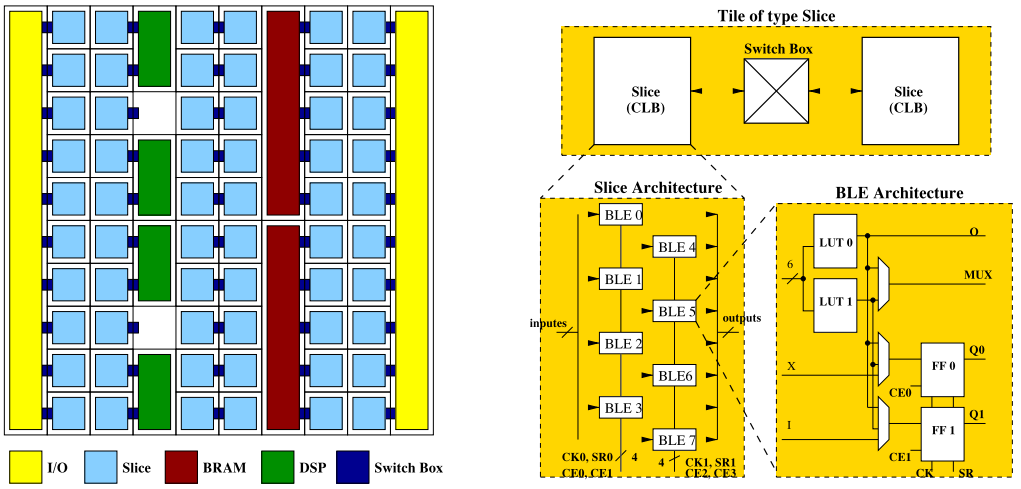
## 2 LITERATURE REVIEW

FPGA placement algorithms can be classified into three main categories based on the optimization approach employed: *Analytic*, *Simulated-Annealing*, and *Partitioning* based. Analytic approaches are used extensively today in many academic and industrial frameworks since they produce high quality results compared to partitioning-based approaches and much less running time than simulated annealing. Analytic placement algorithms, like QPF [26], StarPlace [25], HeAP [8], and LLP [22] use smooth functions to approximate a non-smooth wirelength cost function and solve a system of equations using efficient numerical methods. However, analytic placers often have difficulty dealing with hard constraints imposed by the FPGA architecture. Moreover, most analytic FPGA placers [7, 25, 26] consider homogeneous, island-style FPGA architectures. More recently, academic analytic placers for heterogeneous FPGA architectures have started to appear in References [4, 8, 22, 5].

Many of the analytic FPGA placers (e.g., References [8, 22, 5]) employ wirelength models that do not consider the type of segmented-routing architectures used in today's modern FPGA architectures. Two exceptions are References [4, 7], both of which seek to minimize segmented-routing wirelength. None of the FPGA analytic placers mentioned above directly model congestion.

Packing has been previously used in some FPGA flows to address congestion. The approach in Reference [20] employs loose packing to avoid over utilization of *Configurable Logic Blocks (CLBs)*, thus reducing congestion. However, this strategy can rapidly lead to the under utilization of CLBs resulting in the circuit failing to fit on the FPGA. Moreover, the uniform under utilization of CLBs can also lead to an increase in total wirelength and hence congestion. To compensate, Un/DoPack [21] first identifies congested regions, unpacks CLBs in those regions, and then re-packs those CLBs with a reduced cluster size. However, the entire process is extremely expensive, because it requires using the actual router to identify the congested regions of CLBs. Also, Un/DoPack does not handle the congestion that is introduced by "flyover" nets (i.e., nets that do not have any pins in that congested region).

Unlike academic FPGA architectures, commercial FPGAs have more hard constraints that limit and complicate traditional packing algorithms that are primarily based on circuit connectivity. Most analytical placers developed for FPGAs in the past did not consider specific routing congestion issues in their flows, which causes some complication in routing these netlists. However, in the spirit of the ISPD 2016 Routability-Driven FPGA Placement Contest, a few routability-driven analytic placers [12, 13, 17, 18] targeting modern Xilinx UltraScale FPGA architecture are published.




(a) UltraScale Layout.

(b) Tile Architecture of Slice Type.

Fig. 1. Xilinx UltraScale architecture.

## 3 BACKGROUND

In this section, essential background on the Xilinx UltraScale FPGA architecture is given, and the placement problem introduced. This is followed by a brief explanation and contrast of the top three winning placement flows from the ISPD 2016 FPGA contest.

### 3.1 UltraScale FPGA Architecture

Xilinx UltraScale FPGAs [24] consist of heterogeneous programmable logic blocks, such as general logic (i.e., Look-Up Tables (LUTs)), random access memory blocks (i.e., BRAMs), and digital-signal processing blocks (i.e., DSPs), as shown in Figure 1(a). As well, there are prefabricated routing segments of different lengths in both horizontal and vertical directions. Slices contain both LUTs and Flip-Flops (FFs), which, when grouped together, share a single switch for routing. Switch boxes provide both intra- and inter-slice connectivity. As shown in Figure 1(b), slices are placed side-by-side such that each pair of slices share the same switch box in a back-to-back formation. Each slice has eight *Basic Logic Elements (BLEs)*, and each BLE contains a single six-input LUT and two FFs. The six-input LUTs can be configured to implement either one logic function of up to six inputs with one output, or two distinct logic functions of up to five inputs and two separate outputs. To implement two distinct functions in the same LUT, the sum of the distinct inputs of the two functions must not exceed five. There are only two clock (CLK) signals and two set/reset signals per slice. The bottom four BLEs share the same CLK and SET/RESET signals, while the upper four BLEs use the second CLK and SET/RESET signals. Every group of four BLEs is further divided into two subgroups, each of which must share the same signal. These control-set constraints, coupled with the other placement constraints, make the placement problem extremely challenging [27]. Even if all of the architectural constraints are satisfied by the current placement, there is no guarantee that the placement will be routable. Moreover, different placement solutions may require the router to perform more or less work to find a feasible routing solution.

### 3.2 Placement Problem

FPGA placement is the process of mapping the cells in a circuit netlist to sites on an FPGA. A netlist is represented by a hyper-graph $G_h(V, E_h)$, where vertices $V$ is a set of cells (i.e., LUTs, FFs, RAMs,

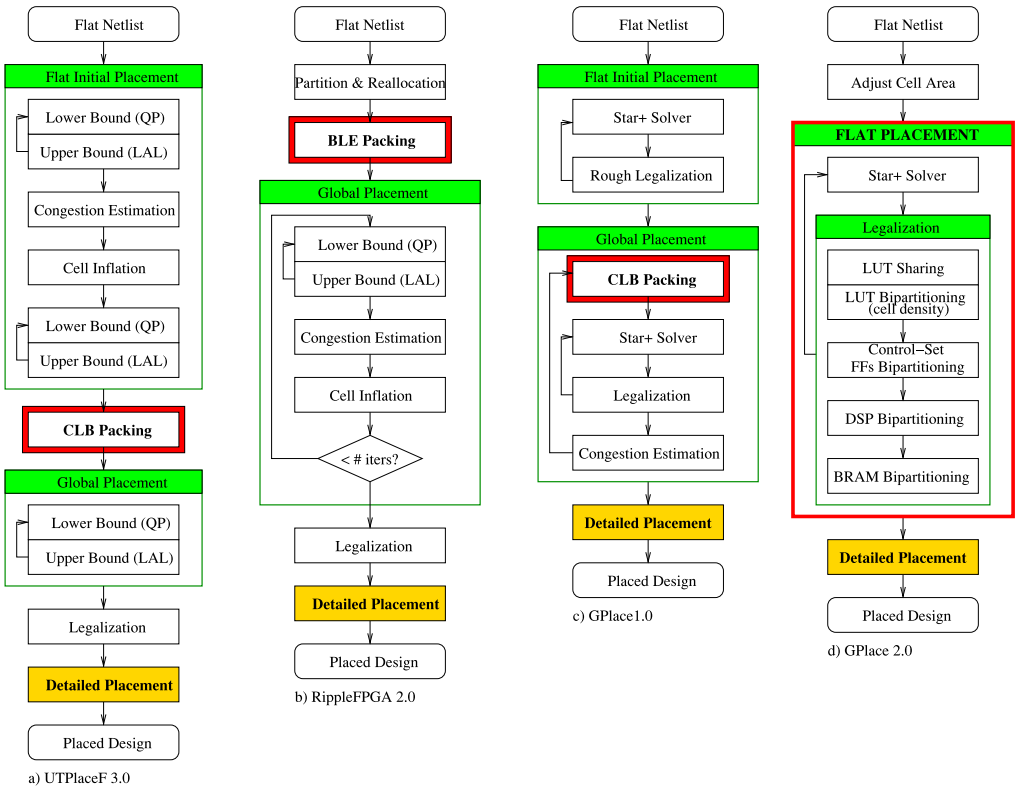



Fig. 2. ISPD 2016 FPGA contest winner flows.

DSPs, and IOs) to be placed and the hyper-edges $E_h$ are a set of nets. Each net is a subset of $V$ that specifies the connections to be made between cells. The architecture defines a rectangular region with width $W$ and height $H$ divided into columns each dedicated to one cell type (i.e., slice, RAMs, DSPs, and IOs), as shown in Figure 1(a). The placement must satisfy all architectural constraints associated with FF control sets, LUT-sharings, block, and column type matching. In routability-driven placement, the goal is to place the cells such that the circuit can be successfully routed and the routed wirelength minimized.

### 3.3 ISPD 2016 Contest Winning Placers

Since 2005, ISPD has held many different contests on topics covering placement, global routing, gate sizing, and clock tree synthesis on ASIC design tools. The ISPD 2016 contest was the first contest on FPGA CAD tools, covering challenging topics related to FPGA CAD Routability-Driven FPGA Placement. The contest benchmarks are based on an industry leading 20nm technology Virtex UltraScale device, and reflect typical modern, high-end FPGA designs [27]. The contest evaluation metrics include wirelength, routability, and runtime. The flows of the top three winning placers of the ISPD FPGA placement contest are shown in Figure 2. Figure 2(a) and (b) show the flows for the first- and second-place contest winners, respectively: UTPlaceF ICCAD16 [13] and RippleFPGA ICCAD16 [18]. Figure 2(c) shows our previous flow, GPlace1.0, which placed third in the contest. Figure 2(d) shows an enhancement of GPlace1.0, called GPlace2.0, that was presented in Reference [17]. Table 1 summarizes the main differences between the contest winners. UTPlaceF

Table 1. ISPD 2016 Placement Contest Winners Flow Comparisons

| FPGA Placer | | | | |
|---|---|---|---|---|
| Placement Module | UTPlaceF ICCAD16 [13] | RippleFPGA ICCAD16 [18] | GPlace1.0 | GPlace2.0 [17] |
| AP Net-Length Model | B2B | B2B | Star+ | Star+ |
| Congestion Estimation | Adaptive ASIC Global Router | WLPA | WLPA | - |
| Congestion Removal | Congestion-Aware Packing | Cell Inflation | Re-Packing | - |
| Global Placement Target | CLB Packed Netlist | BLE Packed Netlist | CLB Packed Netlist | Flat Netlist |
| Detailed Placement | Congestion-Aware ISM | Optimal Region Greedy Moves | Local Greedy Moves | Local Greedy Moves |

ICCAD16 and RippleFPGA ICCAD16 both use an *Analytical Placement (AP) Bound-to-Bound (B2B)* [19] net length model, while GPlace1.0 and GPlace2.0 employ the same wirelength model used in StarPlace [25], a near-linear model, called Star+. To estimate congestion during the global placement, RippleFPGA ICCAD16 and GPlace1.0 use a probabilistic method based on *Wirelength Per Area (WLPA)* [17, 18], whereas UTPlaceF ICCAD16 uses an ASIC global router (NCTUgr2.0) [14]. Since the NCTUgr2.0 global router works on a grid graph and does not handle the different segment lengths present in modern FPGA architectures, UTPlaceF ICCAD16 experimentally adjusts channel widths in the grid graph to produce a more accurate congestion map.

In the global placement stage, UTPlaceF ICCAD16, RippleFPGA ICCAD16, and GPlace1.0 optimize wirelength for packed netlists; however, GPlace2.0 performs optimization on a *flat* netlist. Both UTPlaceF ICCAD16 and GPlace1.0 rely on packing to reduce the routing demand and hence congestion, while RippleFPGA ICCAD16 uses BLE bloating to reduce the congestion. Traditionally, alleviating congestion in FPGAs is addressed by packing to produce a placement-friendly netlist. However, packing approaches may unnecessarily restrict the placement solution space leading to degraded wirelength and possibly unroutable placements.

During the final detailed placement stage, UTPlaceF ICCAD16 further minimizes *Half-Perimeter Wirelength (HPWL)* using an ISM approach [13], while RippleFPGA ICCAD16 uses the *Optimal Region Greedy Moves (ORGM)* to optimize HPWL. Both GPlace1.0 and GPlace2.0 use local greedy refinements to further optimize the Star+ cost. Below, we summarize some key issues in existing placers that motivate our work in this article.

- —Logical packing algorithms, such as those proposed in References [20, 21], perform packing based only on logical connectivity, which could cluster cells that are physically far apart. This leads to wirelength-unfriendly netlists that may degrade routability.
- —Placement-aware packing techniques (e.g., UTPlaceF ICCAD16, and GPlace1.0), which consider physical locations of cells during packing, may still produce netlists that are wirelength-unfriendly. These placers perform packing based on physical locations generated by first performing a *Flat Initial Placement (FIP)*. The FIP does not consider control set constraints associated with the FFs. However, designs with large numbers of control sets (e.g., clocks, resets and control enables) are common in modern high-end FPGAs. This reduces the logic utilization and hence forces the packing method to pack FFs that are far apart to fit into the FPGA area, possibly degrading wirelength and routability. This issue will be discussed in the experimental results section, where we compare UTPlaceF ICCAD16 and GPlace3.0.
- —Even though RippleFPGA ICCAD16 does not fully pack the netlist, its global placement phase does not handle control set constraints appropriately. This leads to a mismatch between the results obtained from global placement and the final legalized results when designs have large numbers of control sets and FFs. Therefore, it is of particular importance to handle these control set constraints during global placement.

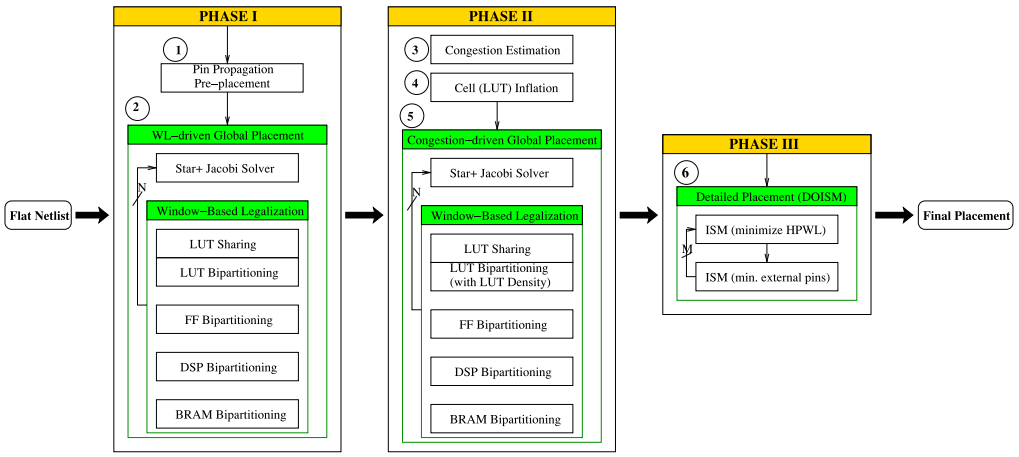



Fig. 3. GPlace3.0 flow.

—The existing flat placement flow in GPlace2.0 is unaware of congestion, and lacks a more efficient detailed placement stage that could further optimize wirelength and improve routability.

Based on the limitations of the current flows discussed above, and the results that will be presented in Section 5, in this work the authors will first attempt to address these limitations by proposing several novel techniques that overcome the drawbacks. The proposed techniques implemented in GPlace3.0 are capable of efficiently resolving congestion, thus improving routability and reducing wirelength. Moreover, another important objective in this work is to develop a placer that is also capable of increasing the chance of routability and reducing the effort of the router to produce solutions in reasonable time. In Section 4, we introduce our methodology for developing GPlace3.0, and highlight the effectiveness of the flow.

## 4 GPLACE3.0 FRAMEWORK

To avoid issues associated with packing that can sometimes restrict the placement solution space, we choose to build upon our previous placer, GPlace2.0 [17]. The proposed flat analytic placer in this article, GPlace3.0, handles FF control-set constraints, as well as LUT sharing during global placement. This eliminates the need for a packing step in the FPGA flow, and allows the placer to optimize globally while enforcing legalization constraints. Since GPlace2.0 [17] does not directly target congestion, a global router was developed for use in GPlace3.0 that accurately estimates congestion for UltraScale FPGA devices. An enhanced cell inflation technique [18] is also proposed to spread cells in congested regions to reduce congestion. Finally, a novel detailed placement stage is added to further improve wirelength. Since GPlace3.0 targets flat netlists (i.e., it does not perform any kind of packing), a *Dual Objective Independent Set Matching (DOISM)* detailed placement is proposed to further improve both wirelength and routability. DOISM alternates between minimizing wirelength and minimizing external pin count (note: external pins are exposed pins that connect different slices, hence, they do not include pins within the same slice).

Figure 3 shows the flow of GPlace3.0. The overall flow consists of three main phases. Phase I seeks to produce a high-quality wirelength placement. This is achieved by performing a pin-propagation pre-placement (Figure 3: step (1)) to place cells close to their IOs. Then, several iterations of analytic flat placement are performed to globally optimize for wirelength while

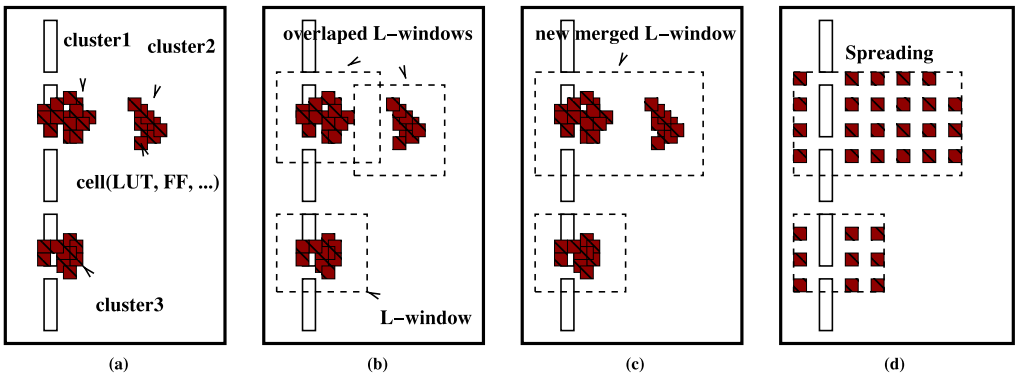



Fig. 4. Window-based legalization flow.

enforcing all legalization constraints. WL-driven global placement (Figure 3: step (2)) in GPlace3.0 is based on the Star+ [25] wirelength model, with extended window-based bi-partitioning legalization to support heterogeneous architectures. The generated placement satisfies all hard constraints and is optimized for wirelength, but still does not, yet, consider congestion. In Phase II, efficient global routing is performed to produce an accurate congestion map (Figure 3: step (3)) and LUTs are inflated (Figure 3: step (4)) (LUT inflation will be explained in Section 4.4) proportional to their number of pins and the congestion values of their tiles. Since GPlace3.0 estimates congestion and inflates cells only once, having an accurate congestion map is crucial to our flow. The congestion-driven global placement (Figure 3: step (5)) shares the same framework with the WL-driven global placement except that inflated LUTs are bi-partitioned based on their densities. In Phase III, detailed placement is performed using DOISM (Figure 3: step (6)) to further optimize for both wirelength and external pin count (to improve routability). In the following subsections, each component of the GPlace3.0 framework is presented in detail, and the main merits of adding each component to the flow is discussed.

### 4.1 [Phase I] Pin Propagation Pre-Placement

An initial placement is generated by propagating the positions of fixed I/O cells through the netlist. First, the netlist is converted to a graph by creating a directed edge from the driver of the net to each sink in the net. The cells are iterated in topological order and moved to the average position of their inputs, placing cells near the input pads they are most connected to. The entire process is repeated, but with the direction of the edges reversed, thereby placing cells near their ultimate outputs. Finally, the results of propagating from inputs and outputs are averaged and cells placed accordingly. Based on our previous work [25], the Star+ analytic technique can benefit from pre-placement; thus producing better results than random initial pre-placement.

### 4.2 [Phase I] WL-driven Global Placement

WL-driven global placement in GPlace3.0 employs the Star+ [25] wirelength model. However, after solving the resulting equation system for wirelength, the placement has real-valued coordinates that (likely) violate the constraints imposed by the architecture, thus requiring legalization. Rather than performing a simple recursive bi-partitioning on the entire FPGA, like in Reference [25], multiple smaller windows are used for bi-partitioning to minimize the displacement between the pre-legalization and post-legalization of cell locations producing a more uniform spreading of cells throughout the placement area. Figure 4 shows the main steps. Adjacent

cells (which could be in the form of LUTs, FFs, DSPs, and BRAMs) that are overutilized (i.e., illegal) are grouped together to form clusters (Figure 4(a)). Clusters are formed using a recursive search that seeks to combine cells in the same site or adjacent sites. For each new cluster, a minimum area window, called a *Legal-window (L-window)*, is determined. The L-window for a cluster must contain sufficient sites to place all of the FFs and LUTs contained in the cluster. Moreover, it must also contain sufficient sites to place any BRAMs or DSPs in the cluster. This ensures that all of the cells within a cluster can be legalized. If multiple L-windows for different clusters overlap, as shown in Figure 4(b), cells belonging to the overlapping clusters are merged into a new cluster, and a new L-window is determined (Figure 4(c)). The cells in each cluster are then legalized within their L-windows using a recursive bi-partitioning procedure that handles FF and control-set constraints as well as LUT sharing (Figure 4). This process removes the need to perform packing.

*4.2.1 Find Minimum L-Windows.* The L-window corresponding to each cluster must contain sufficient sites to legalize all the FFs, LUTs, DSPs, and BRAMs contained by the cluster. The number of sites required to completely legalize shared LUTs, BRAMs, and DSPs can be computed in O(1) based on their numbers. However, calculating the number of sites required to legalize FFs with control-set constraints takes linear time complexity, as explained later in Section 4.2.3. The pseudo-code for finding a minimum L-window for each cluster is summarized in Algorithm 1. The process of finding a minimum L-window occurs in three stages. Initially, the L-window is set to the bounding box of the cluster (line 1). The first stage (lines 2–4) then expands the bounding box by doubling the smaller dimension of the L-window until the L-window is large enough to contain all of the cells in the cluster. When doubling a dimension, the L-window is automatically cropped, as necessary, so as to not exceed the dimensions of the FPGA. If the smaller dimension is already as large as it can be (i.e., the L-window already spans the full width of the FPGA but not the height), the larger dimension is doubled instead. The second stage (lines 5–20) performs a binary search to narrow down the approximate minimum dimensions of the L-window. The L-window is only modified on the larger of the two axes. The initial search space (lines 6–7) ranges from 0 to the current dimension of the L-window on the larger axis. The final stage (lines 21–32) crops the edges of the L-window by a single row/column at a time. The complexity of finding an exact minimum L-window is $O(log(n))$, where $n$ is the size of the larger dimension of the final L-window.

After each cluster's L-window has been determined, it is possible that some of the L-windows overlap. Since each cluster assumes that it has access to all of the resources in its L-window, this can result in insufficient resources to legalize a second cluster in the case of two overlapping clusters. Therefore, before legalizing within an L-window, any clusters whose L-windows are overlapping are merged, and a new L-window is calculated for the merged cluster. This is repeated until no L-windows overlap. For large benchmarks, the number of final clusters is only 10% of the original clusters identified. Every time two clusters are merged, the required sites must be recalculated before an exact L-window can be computed, which can be fairly costly if many clusters require merging. To minimize the number of clusters requiring merging, an estimation of L-window size is computed in $O(1)$ time using only the number of LUTs. This estimation is used to preemptively merge nearby clusters before any exact L-windows are calculated.

Pseudo-code for performing preemptive merging is shown in Algorithm 2. During preemptive merging, L-window sizes and locations are estimated in $O(1)$ time. This estimation is used to preemptively merge nearby clusters before any exact windows are calculated. The complexity of preemptive merging is $O(n^2)$, where $n$ is the number of clusters. Windows are assumed to be square. Windows are centered on the same point as the bounding box for the cluster. When no two clusters have estimated L-windows that overlap, preemptive merging terminates.

**ALGORITHM 1:** Find Minimum Window

**Require:** Cluster *cluster*: a group of overutilized adjacent cells.
**Ensure:** L-window: a minimum rectangular window centered in the middle of the cluster, and sufficient to contain all cluster's cells.

```
1:  cluster.window ← cluster.boundingBox;
2:  while cluster.window.resources < cluster.resources do
3:      cluster.window.smallerDimension ← cluster.window.smallerDimension * 2;
4:  end while

5:  axis ← cluster.window.largerDimension.axis;
6:  upperBound ← cluster.window[axis].max − cluster.window[axis].min;
7:  lowerBound ← 0;
8:  clusterCenter ← cluster.window[axis].min + (upperBound + lowerBound)/2;
9:  while lowerBound <= upperBound do
10:     testDim ← (upperBound + lowerBound)/2;
11:     cluster.window[axis].min ← clusterCenter − (testDim/2);
12:     cluster.window[axis].max ← clusterCenter + (testDim/2) + (testDim%2);
13:     if cluster.window.resources <= cluster.resources then
14:         upperBound ← testDim − 1;
15:     else
16:         lowerBound ← testDim + 1;
17:     end if
18: end while
19: cluster.window[axis].min ← clusterCenter − (lowerBound/2);
20: cluster.window[axis].max ← clusterCenter + (lowerBound/2) + (lowerBound%2);

21: while cluster.window is shrinking do
22:     for axis in X, Y do
23:         cluster.window[axis].min ← cluster.window[axis].min + 1;
24:         if cluster.window.resources < cluster.resources then
25:             cluster.window[axis].min ← cluster.window[axis].min − 1;
26:         end if
27:         cluster.window[axis].max ← cluster.window[axis].max − 1;
28:         if cluster.window.resources < cluster.resources then
29:             cluster.window[axis].max ← cluster.window[axis].max + 1;
30:         end if
31:     end for
32: end while
```

*4.2.2 LUT Sharing in Bi-partitioning.* The goal of LUT-sharing is to reduce LUT utilization (i.e., number of six-input LUTs) throughout the FPGA by seeking opportunities to combine LUTs. A slice in the UltraScale architecture contains eight six-input LUTs. However, a single six-input LUT can implement two LUTs if together the number of unique inputs is less than six. GPlace3.0 pairs LUTs while aiming to maximize the number of shared inputs and minimize the distance between the paired LUTs in the placement. First, GPlace3.0 searches for groups of LUTs that have $n$ common inputs. LUTs with common inputs are found by hashing all possible subsets of their inputs of size $n$ as in Reference [17]; that is, for sharing $n$ inputs, a $k$-input LUT hashes all ($k$ choose $n$) subsets of its inputs and is added to the corresponding buckets. After all LUTs have been hashed, each bucket is partitioned into groups that share the same $n$ inputs. Next, each LUT is paired to the

**ALGORITHM 2:** Preemptive Merging

**Require:** $clusters[1..n]$;
**Ensure:** $clusters[1..m], m \leq n$ list of clusters after merging overlapped clusters by fast guessing of their L-window sizes;

```
1: while true do
2:   merges ← 0;
3:   for i ← 0 to i ← n do
4:     for j ← i + 1 to j ← n do
5:       for cluster in {clusters[i],clusters[j]} do
6:         center ← center(cluster.boundingBox);
7:         sidelength ← sqrt(cluster.numLUTs/8);
8:         window[i] ← square(center, sidelength);
9:       end for
10:      if overlap(window[i], window[j]) then
11:        merge(cluster[i], cluster[j]);
12:        merges ← merges + 1;
13:      end if
14:    end for
15:  end for
16:  if merges == 0 then
17:    break;
18:  end if
19: end while
```

nearest unpaired LUT in the same partition, provided the distance between them is less than 1 in the placement. The pairing process is repeated for all values of $n$, starting at the highest number of shared inputs (five) and ending with the fewest (zero).

The dependence between the sharing of LUTs and the LUT cell capacity of a slice makes identifying overflowing regions difficult. A slice may contain 8 six-input LUTs; however, if the LUTs are smaller and can thus share, a slice can contain up to 16 LUTs. To resolve this dependence, GPlace3.0 pairs LUTs prior to legalization and removes one LUT of each pair from the placement. Removing one LUT from each pair allows bi-partitioning to use a capacity of eight LUTs per slice. After legalization, the removed LUTs are added back to the placement in the same position as their paired LUT.

*4.2.3 Flip Flop Control Sets in Bi-partitioning.* A slice may contain 16 FFs provided that they satisfy several constraints on their clock, CE, and reset signals. The 16 FFs form a hierarchy based on their control signals. At the top level, the 16 FFs can be subdivided into two groups of 8, with each group sharing the same clock and reset signal. Next, each group of eight must be subdivided into two groups of four, each group sharing the same CE signal. The bi-partitioning algorithm requires that the number of slices needed for all FFs in a region be computed and updated as FFs move between regions. To compute the slice count, GPlace3.0 maintains a tree structure organizing the FFs in the circuit during partitioning. An example of tree structure to compute the required number of slices for FFs in a region is shown in Figure 5. The root of the tree has a child for each region in the tree used for bi-partitioning. Figure 5 shows only Region 0 (i.e., RG0). Each region node has a child for each clock and reset combination (i.e., CR# in the region). In the next level, the clock and reset nodes have a child for each CE (e.g., CE#). The next level contains a count of the number of FFs that match the CE, reset, clock, and region as its nodes branch in the tree. Computing the number of slices required in each region is carried out in a bottom-up pass through the tree. At

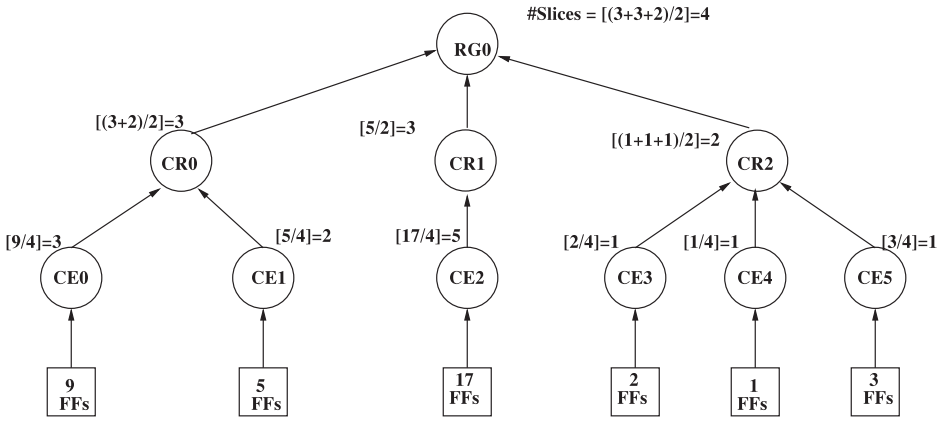



Fig. 5. Control signal FF bi-partition tree.

the bottom, the CE node returns the number of groups of four it will require (⌈FF/4⌉). The clock and reset node sums the value from each child and returns ⌈sum/2⌉, thereby computing the number of half slices. Finally, the region node sums the half slice values from each child and returns ⌈sum/2⌉, thereby computing the number of slices required in the region. This tree is used to identify overflow in a region by comparing the number of slices required to legalize FFs in the region with the available slices. Tree branches are updated when moving a FF between regions in $O(1)$ time (the height of the tree is always 3). Figure 5 shows that region RG0 contains 37 FFs with different clock, reset, and CE signals. If partitioning FFs considers only the capacity of a slice (i.e., 16 FFs per slice), the minimum number of slices required to legalize FFs is three. However, to enforce all control-set constraints, the exact number of slices required to legalize FFs in RG0 is four, as shown in Figure 5.

### 4.3 [Phase II] Congestion Estimation using a Global Router

The main goal of Phase I is to produce an initial placement that minimizes wirelength and routing demand. However, there is no guarantee that the resulting placement is free of congestion or that it can be even routed. Therefore, the objective of Phase II is to further reduce congestion to increase the likelihood of producing a routable placement without significantly degrading wirelength. The basic idea is to first use a global router once to estimate and identify regions of congestion (as explained in Section 4.3.2), and then perform cell (i.e., LUT) inflation to resolve congestion, which is presented in Section 4.4. Since congestion mitigation is only performed once, it is imperative that the global router produces an accurate congestion estimate. Therefore, we have made modifications to the PathFinder Global Router [15] to obtain a more accurate congestion estimate. Further motivation and details are given in the following subsections.

*4.3.1 The Need for Accurate Congestion Estimation.* Failure to accurately identify and estimate congested regions may lead to two equally undesirable situations: (1) unnecessary inflation of uncongested cells resulting in excess wirelength and worse routability or (2) less application of inflation to congested cells, thus keeping the problem of congestion unresolved. Since congestion estimation is only performed once in Phase II, having an accurate congestion estimation technique is crucial. To address this, a modified *PathFinder Global Router (mPFGR)* was developed to accurately estimate congestion. The proposed modifications encompass three important features. The first involves adding an adaptive overflow penalty based on the router's progress to the original, negotiation-based cost function presented in NCTUgr2.0 [14]. This adaptive penalty enables the global router to converge much faster and with better wirelength compared to the original

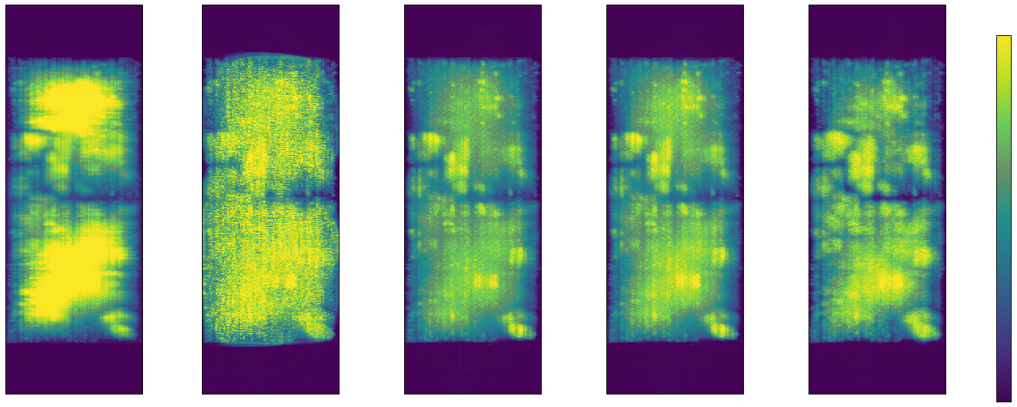



Fig. 6. Congestion Map of FPGA05 in ISPD2016 contest: (a) WLPA congestion estimation; (b) PFGR (3 iters.) using VPR-PF costs; (c) NCTUgr2.0 cost; (d) mPFGR (3 iters.) using modified cost; (e) Routing congestion by Vivado detailed router.

Table 2. Comparison between Different Congestion Estimation Methods

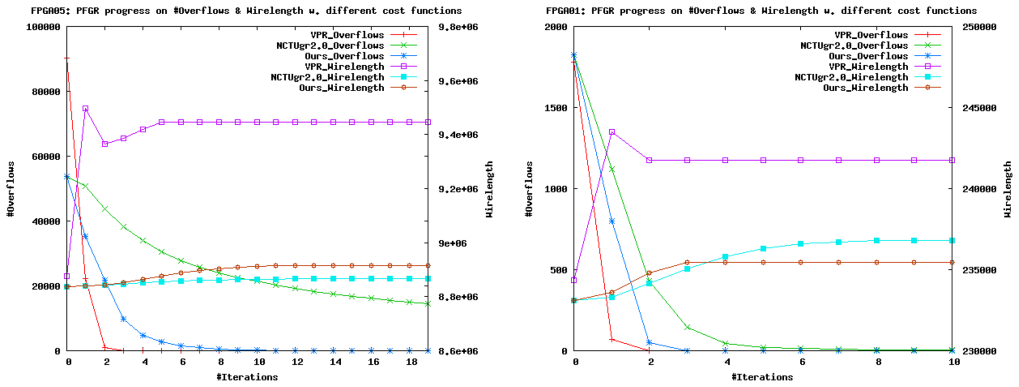
| Benchmark | Metric | WLPA | VPR-PF cost | NCTUgr2.0 cost | mPFGR cost |
|---|---|---|---|---|---|
| FPGA05 | SAD[a] | +8778.4 | +9705.8 | +3683.4 | **+3591.4** |
| | AANE[b] | 0.0976 | 0.1079 | 0.041 | **0.039** |
| FPGA03 | SAD | +5014.7 | +2756.6 | +2258.6 | **+2204.3** |
| | AANE | 0.0625 | 0.0343 | 0.0281 | **0.0275** |

[a]SAD: Sum of absolute difference.
[b]AANE: Average absolute normalized error.

cost function. The second modification introduces two simple heuristic methods to capture the effects of local nets based on the number of pins in the switch box. The goal here is to close the performance gap between the global router and detailed router by taking into account the contribution of local wirelength and routing, respectively, when estimating global congestion. The third modification involves using an A* search to speedup the maze router employed in mPFGR.

Before presenting the algorithmic details of the previous modifications, we provide the reader with some preliminary visual evidence in the form of congestion heat maps for the effectiveness of mPFGR versus other, well-known congestion-estimation methods. Figure 6 shows the congestion heat maps obtained using different techniques including WPLA, Versatile-Place-and-Route's cost function augmented with a PathFinder congestion estimate (VPR-PF), PathFinder congestion based on NCTUgr2.0 cost, mPFGR proposed in this work, and the Vivado detailed router. It is important to emphasize that although the congestion heat maps are for a single ISPD 2016 contest benchmark (i.e., FPGA05), the pictures they give are very typical for congested placements.

In addition to a visual comparison, the *Sum of Absolute Difference (SAD)* and *Average Absolute Normalized Error (AANE)* measures from [28] can be used to numerically measure the difference between congestion heat maps. Table 2 shows the SAD and AANE between the different estimation methods relative to the actual congestion after the Vivado detailed router is applied (for two ISPD 2016 contest benchmarks). Based on lower values for SAD and AANE, the proposed mPFGR implementation shows superiority over the other methods.




(a) (b)

Fig. 7. mPFGR with different cost functions: (a) On FPGA05 and (b) On FPGA01.

Table 3. PathFinder Global Router Parameter Definitions

| Parameter | Definition |
|---|---|
| $p_r$ | present congestion penalty |
| $h_r$ | historical congestion penalty |
| $base_r$ | base cost |
| $b_r$ | Manhattan distance between the origin and the destination switches |
| $cap_r$ | capacity of a given routing resource |
| $dem_r$ | demand on a given routing resource |

*4.3.2 Modifications to PathFinder.* Negotiation-based routers require signals to negotiate with each other to obtain routing resources. ASIC global routers, such as NCTUgr2.0 [14], BFR [11], and FastRoute [16], use different negotiation-based cost functions to overcome the drawbacks that are present in the original PathFinder global router. For example, NCTUgr2.0, a state-of-the-art ASIC global router, employs an adaptive *historical congestion* cost that can decay over time as edges in the (grid) graph become less congested. However, NCTUgr2.0's *overflow* penalty function is not adaptive. Consequently, the router can take a long time to resolve overflows and can converge to a suboptimal solution. This situation is magnified in highly congested designs. For example, in Figure 7(a), the number of overflows for NCTUgr2.0 decreases as the number of iterations increases, but NCTUgr2.0 prematurely converges without resolving a large number of overflows. BFR [11] seeks to avoid the previous problem by adaptively increasing the overflow penalty as the router progresses. However, the overflow penalty increases by a predetermined amount over time. Thus, circuits that require different amounts of time to route may be assigned a very different penalty at the same relative point in the routing process. Unlike BFR [11], mPFGR seeks to adaptively increase the magnitude of the overflow penalty based on the on-going progress of the router. Specifically, the penalty assigned is based on the ratio of the initial and current overflows and, therefore, takes into account the progress of the router for a particular design.

The PathFinder cost functions used by mPFGR were adapted from NCTUgr2.0 [14]. Equation (1) is used to determine the cost of using a global routing resource [15]. Table 3 shows the definition of the parameters presented in Pathfinder's cost functions. Equation (2) is the sum of the overflows of a global routing resource and is updated at the end of each PathFinder iteration. Equation (3)

represents the historical congestion penalty for a global routing resource. Its value depends on the current PathFinder iteration and the value of Equation (2). $C_1$ is a constant that scales the historical congestion penalty and $C_2$ is a constant that causes the historical congestion penalty to decay based on the number of PathFinder iterations. In our implementation, $C_1$ is set to 7.0 and $C_2$ is set to 4.0, similar to Reference [14]:

$$cost_r(i) = (1 + dah_r(i)) \times p_r(i) + base_r(i), \tag{1}$$

$$h_r(i) = \begin{cases} h_r(i-1) & \text{if } cap(r) \geq dem(r) \\ h_r(i-1) + (dem(r) - cap(r)) & \text{otherwise} \end{cases}, \tag{2}$$

$$dah_r(i) = \frac{h_r(i)}{C_1 + C_2 \times \sqrt{i}}. \tag{3}$$

Equation (4) represents the present congestion penalty for a global routing resource. Its value depends on the demand and capacity of a global routing resource, the ratio of the number of overlaps in the first PathFinder iteration to the most recent PathFinder iteration, and the current PathFinder iteration:

$$p_r(i) = \begin{cases} 1 + \frac{C_3}{1+e^{C_4(cap(r)-dem(r))}} & \text{if } cap(r) > dem(r) \\ 1 + \frac{C_3}{1+e^{C_4(cap(r)-dem(r))}} + (1 + log(\frac{\text{initial overlaps}}{\text{current overlaps}})) \times 20 \times i & \text{otherwise} \end{cases}, \tag{4}$$

$$base_r(i) = (C_5 + C_6/2^i) \times b_r. \tag{5}$$

In our implementation, $C_3$ is set to 150 and $C_4$ is set to 0.3 similar to Reference [14]. However, Equation (4) differs from the cost functions used by NCTU-GR 2.0 as a discrete jump in cost is added when the demand of a global routing resource is greater than its capacity. The magnitude of this penalty is determined based on the current PathFinder iteration and the ratio of the number of overlaps in the initial PathFinder iteration to the number of overlaps in the most recent PathFinder iteration. This penalizes global routing resources more heavily if they remain congested during later stages of the route. Equation (5) represents the base cost of using a global routing resource. This term decreases for each PathFinder iteration, which decreases the wirelength portion of the cost for each routing resource relative to the overflow/congestion portion of the cost. This contributes to the global router prioritizing wirelength more early in the route and overflow reduction later in the route. In our implementation $C_5$ is set to 30 and $C_6$ is set to 200, similar to Reference [14]. This function differs from the cost function used by NCTUgr2.0 in that the term $C_5 + C_6/2^i$ is multiplied by $b_r$, which is the Manhattan distance between the origin switch and the destination switch of the global routing resource. This is because NCTUgr2.0 is an ASIC global router that operates on a grid graph, and is not segmented like the global routing architecture of some FPGAs. Figure 7(a) and 7(b) shows that our new cost resolves overflows much sooner than the cost functions used in NCTUgr2.0 [14]. To capture the effects of local nets on the global congestion map, two simple heuristic methods are used in mPFGR. The first represents local wirelength based on the number of pins located in a site connected to a switch, and involves adding demands to edges that are connected to the switch box based on Equation (6). $K_1$ is equal to 0.0008 in our experiments:

$$dem(e) = dem(e) + (K_1 \times \#pins \times cap(e)). \tag{6}$$

The second heuristic is used to reduce the capacity of the switch boxes by a value proportional to the number of pins located in a site connected to that switch. Since local connections within a tile are not a full cross bar, some of these local nets should be routed globally, outside of the tile. This correction method is inexpensive to calculate, and captures the effect of the local routing by

**ALGORITHM 3:** PathFinder Global Router (mPFGR)

**Require:** Netlist, placement and global routing resources graph for device
$RT[n] \leftarrow \varnothing \quad \forall n \in$ netlist
$h_c[r] \leftarrow 1 \quad \forall r \in$ global routing resources
**do**
  **for** each net $i \in$ netlist **do**
    $RT[net_i] \leftarrow \varnothing$
    **for** $\{1, \ldots, |net\_sink\_switches_i|\}$ **do**
      $prev\_rr[s] \leftarrow NULL$, $dist[s] \leftarrow \infty$, $visited[s] \leftarrow false \quad \forall s \in$ switches
      $PQ \leftarrow \varnothing$
      $dist[origin\_switch$ of $net\_driver_i] \leftarrow 0$
      Push $origin\_switch$ of $net\_driver_i$ into $PQ$ with value 0
      $dist[destination\_switch$ of $r] \leftarrow 0 \quad \forall r \in RT[net_i]$
      Push $destination\_switch$ of $r$ into $PQ$ with value 0 $\quad \forall r \in RT[net_i]$

      **while** $PQ \neq \varnothing$ **do**
        $current \leftarrow$ pop top of $PQ$
        $visited[current] \leftarrow true$
        **if** $current \in net\_sink\_switches_i$ and sink not found **then**
          Mark sink as found
          $traceback\_switch \leftarrow current$
          **while** $prev\_rr[traceback\_switch] \neq NULL$ **do**
            $RT[net_i] \leftarrow RT[net_i] \cup prev\_rr[traceback\_switch]$
            $traceback\_switch \leftarrow origin\_switch$ of $prev\_rr[traceback\_switch]$
          **end while**
          **break**
        **else**
          **for** each outgoing global routing resource $r$ of $current$ **do**
            $dest\_sw \leftarrow destination\_switch$ of $r$
            $grr\_cost_r \leftarrow (1 + dah_r(i)) \times p_r(i) + base_r(i)$
            $new\_cost \leftarrow dist[current] + grr\_cost_r$
            **if** $visited[dest\_sw] = false$ and $new\_cost < dist[dest\_sw]$ **then**
              $dist[dest\_sw] \leftarrow new\_cost$
              $prev\_rr[dest\_sw] \leftarrow r$
              Push $dest\_sw$ into $PQ$ with value $new\_cost$
            **end if**
          **end for**
        **end if**
      **end while**
    **end for**
  **end for**
  $h_c[r] \leftarrow h_c[r] + min(0, dem[r] - cap[r]) \quad \forall r \in$ global routing resources
**while** overlaps exist

adding blockages to the edges connecting to the switch box based on Equation (7). $K_2$ is 0.0028 in our experiments:

$$cap(e) = cap(e) - (K_2 \times \#pins \times cap(e)). \tag{7}$$

*4.3.3 mPFGR Algorithm: Discussion and Pseudo-Code Explanation.* The pseudo-code of the mPFGR proposed in this work is shown in Algorithm 3. mPFGR uses the PathFinder negotiated-congestion algorithm, with the modifications explained earlier, to route all nets and then iteratively

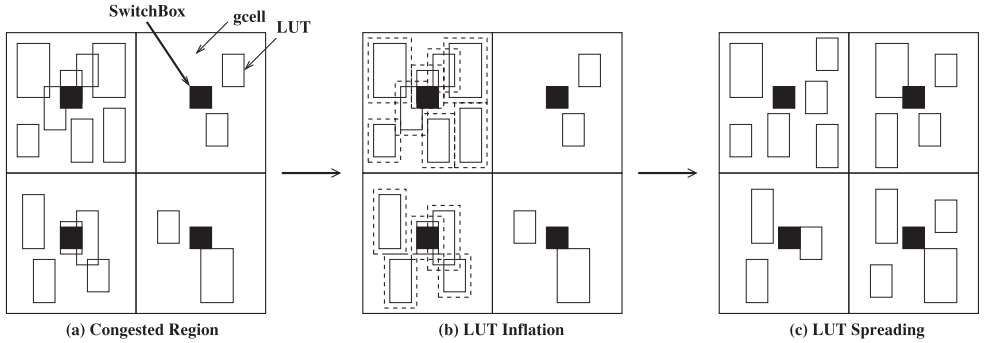



Fig. 8. Resolving congestion via cell inflation.

resolve overlaps. On each PathFinder iteration (lines 4–40), the routing tree of each net is ripped up and then re-routed by invoking a maze router multiple times (lines 5–39). An A* search is used to speedup the maze router. The goal of the maze router is to find the path from a source node to a destination node with minimum cost. The cost of a path is simply the cost of all the routing resources that make up the path. The cost of using a routing resource is defined by the PathFinder cost functions (line 28).

The PathFinder cost functions are then updated based on the current usage of each routing resource, the historical usage of each routing resource, and the current PathFinder iteration. The current usage of each routing resource is updated dynamically based on the route trees of each net, while the historical usage of each routing resource is updated at the end of each PathFinder iteration (line 40). Initially the global router prioritizes minimizing wirelength. The dynamic PathFinder cost functions result in overlap reduction being prioritized more in later iterations.

Each time a net is routed the maze router is invoked $|S_{net}|$ times, where $S_{net}$ is the set of sinks for a net. Each time the maze router is invoked (lines 8–39), the search is initialized with the switches in the current routing tree for the net (lines 9 to 12). Initially the route tree for the net is empty, and the search is initialized with just the switch of the net driver. When a sink is located the current route tree for the net is expanded to include the path to the sink (lines 18–23), and the maze router is restarted. This step is repeated until all sinks are located and the route tree for the net is connected to all of the nets pins.

*4.3.4 Algorithm Complexity.* The maze router is invoked $i_p \times n \times \bar{s}$ times, where $i_p$ is the number of PathFinder iterations required to remove all overlaps, $n$ is the number of nets in the netlist and $\bar{s}$ is the average number of pins per net. The maze router uses breadth first search, which has a worst case time complexity of $O(|V| + |E|)$, where $V$ is the set of switches and $E$ is the set of global routing resources. Thus, the overall time complexity for the global router is $O(i_p \times n \times \bar{s} \times (|V| + |E|))$.

### 4.4 [Phase II] Cell (LUT) Inflation

The first step to relieve and mitigate congestion is addressed by estimating congestion via the proposed Global Router (mPFGR) introduced in the previous section. Resolving congestion involves a second important step in the form of cell/LUT inflation. The basic idea behind cell inflation is to artificially increase the size (i.e., area) of cells found in congested regions, so that fewer of those cells occupy the congested region (i.e., reduce the cell density), thus leading to less congestion. Consequently, this should minimize the local pin density (interconnect density) in the identified

congested area and ultimately improve routability. Figure 8 shows the affect of inflating cells/LUTs to further resolve congestion within GPlace3.0.

The cell-inflation method employed by GPlace3.0 is based on work published in Reference [10] and involves performing the following steps. The cell density is computed for LUTs based on its number of pins, congestion level of the gcell it is located in (as shown in Figure 8), and the congestion level of the circuit/benchmark in total. The density of FFs, RAM, and DSP are not varied. Equations (8) and (9) show how LUTs are inflated based on congestion:

$$\mathrm{S} = \mathrm{C}_1 + \mathrm{Cong_B}.\mathrm{C}_2. \tag{8}$$

The factor $S$ is a scaling factor that modifies the amount of LUT inflation peformed based on the amount of congestion of the benchmark. The constants $C_1 = 0.36, C_2 = 0.28$ are empirically determined while and $Cong_B$ is the average congestion of the top 10% congested switches (e.g., congestion level > .65). Originally, LUT density is set to 1. If the LUT is located withing a congested switch box that has a congestion $\geq 0.5$, then the LUT density is computed as the following:

$$\mathrm{density(LUT)} = 1 + \mathrm{S}.\left(\frac{\mathrm{inputs(LUT)} + \mathrm{outputs(LUT)}}{\mathrm{ratio}.\mu_{LUT}} - 1\right), \tag{9}$$

where $\mu_{LUT}$ is the average number of pins of LUTs in the net-list, and $ratio$ is computed based on the congestion value of the switch that LUT is connected to.

$$ratio = \begin{cases} 0.8 & \text{if } 0.5 \leq cong < 0.675 \\ 0.7 & \text{if } 0.675 \leq cong < 0.85 \\ 0.6 & \text{if } 0.85 \leq cong < 1.025 \\ 0.5 & \text{if } 1.025 \leq cong < 1.2 \\ 0.4 & \text{if } cong \geq 1.2 \end{cases}.$$

Bi-partitioning uses the calculated density of cells to decide if a region is overflowing when moving cells between regions. If both regions are overflowing, the legalization will attempt to evenly distribute the density between regions.

### 4.5 [Phase II] Congestion-driven Global Placement

Following congestion estimation and cell inflation, a second round of flat placement is performed that includes a soft cell-density constraint, with the aim of improving wirelength. This optimization is necessary, as wirelength tends to deteriorate after performing cell inflation and (local) LUT movement. Overall, the congestion-driven global placement flow in Phase II shares the same framework with the WL-driven global placement flow in Phase I, but with the exception that LUT bi-partitioning handles different cell (i.e., LUT) densities during legalization.

*4.5.1 Cell Density in Bi-partitioning.* In the case of cell-density, there are two types of overflow that can occur: *hard* and *soft*. Hard overflows occur when a placement region does not have enough sites to accommodate all of the cells (i.e., LUTs and flip-flops) assigned to that region. In GPlace3.0, hard constraints are strictly avoided by using bi-partitioning to ensure legality. However, soft overflows are permitted. Soft overflows occur when the total cell density exceeds the capacity of the placement region. In the latter case, cell density is kept roughly balanced between the two partitions. Figure 9 provides an illustration of cell-density bipartitioning. For the sake of simplicity, each site in Figure 9 is assumed to have a unit area. Initially, all LUTs are assumed to have an area of 1 (Figure 9(a)). Following LUT inflation, LUTs c and d have their area increased to 2 (Figure 9(b)). During LUT bi-partitioning, two constraints must be satisfied. First, each partition must have enough sites to accommodate the total number of LUTs assigned to the partition. (This is a hard constraint.) Second, the total area of the LUTs assigned to a partition must not exceed the

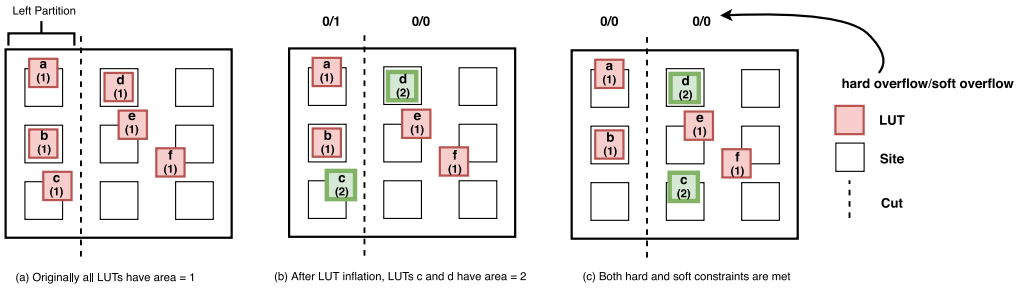
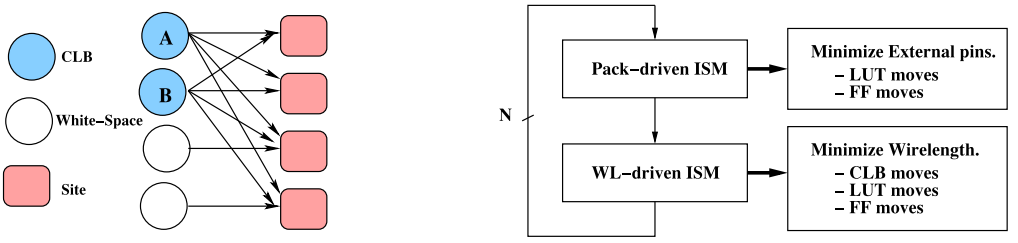



Fig. 9. LUT bipartitioning with cell-density.




(a) ISM bipartite graph.

(b) Dual-Objective Independent Set Matching Flow

Fig. 10. Dual-objective independent set matching.

number of sites in that partition. (This is a soft constraint.) In Figure 9(b), the left partition satisfies the hard constraint, since there are three available sites and three LUTs (i.e., a, b, and c). However, the soft constraint for the (left) partition is not satisfied. This is because the total area of the three LUTs (i.e., a, b, and c) in the (left) partition is 4 (i.e., 1+1+2), and, therefore, exceeds the number of available sites (i.e., 3) in the partition. In this case, the left partition has a soft overflow of 1. Both hard and soft constraints are met for the right partition, since both the number of LUTs (i.e., 3) and the total area of the LUTs (i.e., 4) is less than the number of sites (i.e., 6) in the partition. When performing cell-density bi-partitioning, the goal is to minimize the displacement of the cells from their original positions (calculated by the analytic solver) such that both hard and soft constraints are met in each partition. As shown in Figure 9(c), this can be achieved by selecting LUT c, since it is closer to the cut, and moving it to the right partition. For some partitions, the soft constraints cannot be satisfied. Therefore, the goal is to try and keep the density of the LUTs in each partition (roughly) the same.

### 4.6 [Phase III] Dual-Objective Hierarchical Independent Set Matching

After Phase II, the current placement has much less congestion compared to the original placement produced by Phase I. However, there is still room for further improvement. In Phase III, we employ a detailed placement technique, called DOISM, to reduce both wirelength and external pin count. (Reduction of the latter effectively improves routability, as shown in Reference [3].) DOISM is based on the ISM technique used successfully in UTPlace3.0 [12]. The basic idea behind ISM is to use bipartite matching to improve wirelength. The bipartite graph consists of two disjoint sets: the first contains cells, and the second contains the current locations of the cells as well as a few locations representing white-space, as shown in Figure 10(a). A weighted edge connects each cell in the first set to each location in the second set, where the weight represents the HPWL wirelength between the cell and the location. By finding a minimum weight matching, cells can be moved (locally) to

minimize wirelength. The key concept in ISM involves ensuring that all of the cells in the first set do not share any nets; hence, all of the cells can be either swapped or moved (to white-space) without affecting the wirelength of any of the other cells in the set.

DOISM shares some similarity to the ISM implementation in Reference [12], but differs in its application. Not only is the conventional ISM used to optimize wirelength, it is also used to reduce external pin count to improve routability. As shown in Figure 10(b), DOISM alternates between optimizing wirelength and reducing external pin count. DOISM seeks to reduce the number of external pins by moving FFs and shared LUTs. The ability to work directly with FFs at this stage of the flow is possible because, unlike Reference [12], FFs have not already been permanently assigned to BLEs. This decision improves global placement earlier in the flow by allowing LUTs and FFs to move freely during optimization while avoiding being locked into a slice. DOISM minimizes wirelength by moving CLBs, FFs, and shared LUTs in a hierarchical fashion. Details are given in the following subsection.

*4.6.1 The DOISM Flow.* Algorithm 4 describes DOISM in detail. Note that DOISM can either minimize wirelength or minimize pin count simply by changing the weights in the independent set-matching graph. The main loop (lines 1–14) iterates over every node (e.g., FF, LUT, or CLB) present in the placement, until the cost improvement drops below a required improvement threshold $I_{th}$. For each round of DOISM, a window (local neighborhood) is generated (line 2) by first choosing the node that has participated in DOISM the fewest number of times to act as the center of the window. This window contains all other nodes and white-spaces that are within a distance $D_{is}$ of the chosen (center) node. An independent set is then generated using only the nodes contained within the window (line 3). Each of these nodes, and their locations on the FPGA, are then used to create a bipartite graph (line 4). Edges are added to the bipartite graph connecting each node to each location, and the cost of the edge is calculated (lines 5-8). A min-cost bipartite matching is performed on the graph using the Hungarian method (line 9). The location of each node in the graph is then updated (line 10), and the number of moves are recorded. Once the total number of moves reaches the threshold $F_{cu}$, a congestion estimation step is performed (line 12), if the congestion-aware feature is enabled.

The independent sets used by DOISM are generated in lines 16–33. Initially, every node is considered to be independent. The first loop (lines 18–31) searches each location within the window. The primary search method used in this implementation begins with the node at the center of the window that was chosen earlier (line 2). The algorithm then searches every other location in sequence, starting with those nearest to the center and searching outwards towards the window boundary. This ensures that many small movements, which are more likely to be favorable, are considered. An alternative search method is also included in the implementation that chooses the locations within the window in a random order. At each location there is only one CLB, but there may be multiple nodes when performing DOISM on LUTs or FFs. In the latter cases, a single node is chosen at random from among those present at the location. By limiting DOISM to one node per location at a time, the legality of the placement is guaranteed to be independent of the movement of any other node. This also has the benefit of increasing the solution space by ensuring the maximum number of distinct locations can be matched. The second loop (lines 19–30) attempts to add each node from the location until one is successfully added. If the chosen node is independent of the current set of nodes, it is added to the set at line 21, and then from each node that shares a net with it is marked as no longer being independent (lines 25–27). The set generation terminates once every location in the window has been searched, or the set reaches its maximum size of $N_{is}$ (lines 22–24). The moving costs (which correspond to the edge weights in the bipartite graph) are then determined (lines 35–48). For white-spaces, if the node is currently in a region with congestion greater than $U_{th}$ the

**ALGORITHM 4:** Dual-Objective Independent Set Matching (DOISM)

**Require:** Maximum independent set size $N_{is}$, maximum independent set radius $D_{is}$, congestion threshold $U_{th}$, required improvement threshold $I_{th}$, and congestion update frequency $F_{cu}$

```
while improvement > I_th do
    Select a window W with a radius of D_is nodes
    S ← GenerateIndependentSet(W)
    Add the nodes in S and their locations to bipartite graph g
    for each pair (n, l) with n ∈ node set of g, l ∈ location set of g do
        cost ← GetMovingCost(n, l)
        Add the edge (n, l, cost) to g
    end for
    Perform min-cost bipartite matching on g
    Update the location of each node in g
    if nodes moved since last congestion update > F_cu then
        Update congestion estimates
    end if
end while

function GenerateIndependentSet(W)
indep[n] ← true for all nodes n
for each location l in W do
    for each node and white space n at location l do
        if indep[n] then
            S ← S ∪ {n}
            if |S| ≥ N_is then
                return S
            end if
            for each node m connected to n do
                indep[m] ← false
            end for
            Continue to next location l
        end if
    end for
end for
return S
end function

function GetMovingCost(n, l)
if n is a white space then
    if congestion at the location of n > U_th then
        return ∞
    else
        return 0
    end if
else if congestion at l > U_th then
    return ∞
else if the move would result in an illegal placement then
    return ∞
end if
return change in HPWL of moving n to location l
end function
```

cost is set to infinity, effectively ensuring that the location is not chosen in the final matching. Otherwise, the cost is set to 0 (lines 37–41). For any other node, if the congestion at the destination exceeds the threshold value, the cost is set to infinity (line 43). The cost is also set to infinity (line 45) if a move would result in an illegal placement. This is mainly a concern with FFs, where there are restrictions on the number of different clock, CE, and reset signals available within a CLB. Finally, if none of the previous restrictions prevent a move, the movement cost is calculated (line 47). This cost corresponds to the estimated change in wirelength that would result from the move, and is found by calculating the HPWL of each net connected to the node before and after the move.

*4.6.2 Time Complexity.* The time complexity of DOISM is $O(u(lD_{is}^2 + k^2\frac{u}{v}N_{is}^2 + N_{is}^3))$, where $u$ is the number of nodes (i.e., LUTs, FFs, and CLBs) in the circuit, $v$ is the number of nets, $k$ is the average number of nets connected to each node, and $l$ is the number of nodes (i.e., LUTs, FFs, and CLBs) at each location within $D_{is}$. Each time function *GenerateIndependentSet* is called, the outer loop (lines 18–31) iterates up to $O(D_{is}^2)$ times. The next (inner) loop (lines 19–30) iterates $O(lD_{is}^2)$ times. However, the interior loop (lines 25–27) is only reached up to $N_{is}$ times, so line 26 is executed $O(k^2\frac{u}{v}N_{is})$ times since each node is connected to $k$ nets, and each net has $k\frac{u}{v}$ pins. This gives *GenerateIndependentSet* a time complexity of $O(lD_{is}^2 + k^2\frac{u}{v}N_{is})$. Each call to function *GetMovingCost* has a worst case time complexity of $O(k^2\frac{u}{v})$, which occurs when the HPWL update on line 47 requires a full recalculation. When a full recalculation is not required, the time complexity is $O(1)$. The time complexity of constructing the bipartite graph (lines 3–8) is $O(lD_{is}^2 + k^2\frac{u}{v}N_{is}^2)$. *GenerateIndependentSet* is called once (line 3), and then *GetMovingCost* is performed $N_{is}^2$ times (line 6). The Hungarian algorithm used for bipartite matching (line 9) requires $O(N_{is}^3)$ time. Updating the location of each node (line 10) requires $O(k^2\frac{u}{v}N_{is})$ time, since the HPWL for each net connected to the $N_{is}$ nodes must be updated. Finally, the ISM is performed $O(u)$ times, resulting in the overall time complexity given above.

## 5 EXPERIMENTAL DESIGN

In this section, we present results obtained by GPlace3.0 in terms of solution quality and CPU time. Section 5.1 gives a detailed description of the benchmarks used in this work. Experimental setup is introduced in Section 5.2 and followed by detailed experimental results. First, we justify the different modules proposed in our framework and highlight the contributions of each. In Section 5.3, the effectiveness of the DOISM detailed placer are presented. This is followed by an explanation of the benefits of the window-based legalization (WB-legalization) approach proposed in Section 5.5. The breakdown analysis of the runtime of GPlace3.0 is then introduced in Section 5.6. Next, a detailed comparison with state of the art placers proposed in the literature is discussed in Section 5.7. In section 5.8, the Vivado router runtime on placements produced by GPlace3.0 and other placers is compared. Finally, in Section 5.10, additional experiments are conducted on an extended set of benchmarks (372 total benchmarks) to highlight the robustness and efficiency of GPlace3.0 placer and compare it again with other state-of-the-art FPGA placers.

### 5.1 Benchmarks

GPlace3.0 was tested using all 12 ISPD 2016 contest benchmarks [23] shown in Table 4. GPlace3.0 was also tested using an additional 360 benchmarks provided directly by Xilinx Inc. These benchmarks were created using a netlist-generation tool, GNL [6]. During synthesis, key circuit features were varied among the benchmarks, including the numbers of LUTs, FFs, DSPs, BRAMS, and IOs. Also, different Rent exponents were used to vary interconnection complexity, and the number of control resets was varied to introduce additional constraints on obtaining a legal placement

Table 4. ISPD 2016 Placement Contest Benchmark Statistics

| Benchmark | #LUTs | #FF | #BRAM | #DSP | #CSet[a] | #IO | R.E[b] |
|---|---|---|---|---|---|---|---|
| FPGA-1 | 49K | 55K | 0 | 0 | 12 | 150 | 0.4 |
| FPGA-2 | 98K | 74K | 100 | 100 | 121 | 150 | 0.4 |
| FPGA-3 | 245K | 170K | 600 | 500 | 1281 | 400 | 0.6 |
| FPGA-4 | 245K | 172K | 600 | 500 | 1281 | 400 | 0.7 |
| FPGA-5 | 246K | 174K | 600 | 500 | 1281 | 400 | 0.8 |
| FPGA-6 | 345K | 352K | 1000 | 600 | 2541 | 600 | 0.6 |
| FPGA-7 | 344K | 357K | 1000 | 600 | 2541 | 600 | 0.7 |
| FPGA-8 | 485K | 216K | 600 | 500 | 1281 | 400 | 0.7 |
| FPGA-9 | 486K | 366K | 1000 | 600 | 2541 | 600 | 0.7 |
| FPGA-10 | 346K | 600K | 1000 | 600 | 2541 | 600 | 0.6 |
| FPGA-11 | 467K | 363K | 1000 | 400 | 2091 | 600 | 0.7 |
| FPGA-12 | 488K | 602K | 600 | 500 | 1281 | 400 | 0.6 |

[a]#CSet: Combination of reset and control-enable signals in the benchmark.
[b]Rent Exponent (RE).

Table 5. Range of Key Circuit Features for 372 Benchmarks [9]

| #LUTs | #FF | #BRAM | #DSP | #CSet | #IO | R.E |
|---|---|---|---|---|---|---|
| 44K-518K | 52K-630K | 0-1035 | 0-620 | 11-2684 | 150-600 | 0.4-0.8 |

Table 6. List of Academic Placers for UltraScale FPGAs

| Placer | ISPD 2016 Contest | ICCAD 2016 | Current Revision |
|---|---|---|---|
| uoguelph | GPlace 1.0 | GPlace 2.0 [17] | GPlace 3.0 |
| utexas | UTPlaceF 1.0 | UTPlaceF 2.0 [13] | UTPlaceF 3.0 [12] |
| CUHK | RippleF 1.0 | RippleF 2.0 [18] | |

solution and thus pose significant challenges during placement. Table 5 shows the range of the previous circuit parameters. Each benchmark is described using the Bookshelf format [23].

### 5.2 Experimental Setup

The main advantage of using the benchmarks listed in Table 4 is that they facilitate direct comparison with other state-of-the-art placement flows that use the Bookshelf format listed in Table 6.

All of the placers are compared based on two metrics: solution quality in terms of routed wirelength, and absolute runtime (in seconds). All placement algorithms ran on an Intel (Xeon CPU E3-1270 v5 @@ 3.60GHz) processor equipped with a 16GB memory. The algorithms were developed using C and compiled using gcc 4.4.7 (Red Hat 4.4.7-18) compiler. The routing is performed by Xilinx Vivado v2015.4 with a patch applied to make it compatible with the modified bookshelf benchmarks format of the ISPD 2016 placement contest. The target device is xcvu95, part of the Virtex UltraScale family. The device's aspect ratio and the average spacing between blocks were determined based on the Xilinx UltraScale VU095 architecture, the latest 20nm FPGA chip. All placers were executed with a single thread and Vivado Xilinx was set to be in the same configurations as in the ISPD 2016 Placement Contest, which limits the running time to 12 hours.

Table 7. Comparison between the Dual-Objective Interleaving ISM (DOISM) and ISM in Reference [12]

| | ISM [12] | | | | DOISM | | | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | Ext. pins[a] | Ext. nets[b] | HPWL | Routed WL | Ext. pins | Ext. nets | HPWL | Routed WL |
| FPGA-1 | 198083 | 64698 | 294219 | 379016 | **172728** | **55301** | **291087** | **355720** |
| FPGA-2 | 345592 | 104505 | 666554 | 661668 | **297309** | **87404** | **660105** | **644438** |
| FPGA-3 | 1045500 | 276455 | 2628590 | 3181163 | **924959** | **236509** | **2604281** | **3101107** |
| FPGA-4 | 1198410 | 300478 | 5203390 | 5528940 | **1055940** | **258181** | **5148517** | **5402715** |
| FPGA-5 | 1393180 | 328140 | 10635200 | 10861993 | **1219330** | **282078** | **10294775** | **10507039** |
| FPGA-6 | 1724060 | 483992 | 4467140 | 6007099 | **1543480** | **425564** | **4411627** | **5820321** |
| FPGA-7 | 1948990 | 522744 | 8297760 | 9755853 | **1727360** | **453892** | **8209415** | **9508871** |
| FPGA-8 | 1805230 | 459024 | 7704620 | 8287680 | **1581590** | **387828** | **7642291** | **8126475** |
| FPGA-9 | 2203190 | 568697 | 10023400 | 11988132 | **1965080** | **493194** | **9940218** | **11710738** |
| FPGA-10 | 1895810 | 618329 | 4867570 | 7194330 | **1684840** | **541593** | **4807377** | **6836300** |
| FPGA-11 | 2030210 | 532315 | 9540550 | 10520913 | **1778600** | **456910** | **9403768** | **10260312** |
| FPGA-12 | 1929700 | 599468 | 5561020 | 7514971 | **1662010** | **502354** | **5479660** | **7224371** |
| Norm. | **+0.00%** | **+0.00%** | **+0.00%** | **+0.00%** | **−11.88%** | **−13.95%** | **−1.43%** | **−2.91%** |

[a]Ext. pins: refer to exposed pins of Slices (CLBs) and ignores absorbed pins within a Slice (CLB).
[b]Ext. nets: refer to exposed nets of Slices (CLBs) and ignores absorbed nets within a Slice (CLB).

### 5.3 Dual-Objective Interleaving Independent Set Matching (DOISM) Effectiveness Validation

To demonstrate the effectiveness of the proposed dual-objective hierarchical ISM (i.e., DOISM), two experiments are conducted. First, DOISM is compared with the ISM proposed in Reference [12] in terms of external pins, external nets, HPWL, and routed wirelength. Table 7 clearly shows that the proposed DOISM results in reductions of −11.88%, −13.95%, −1.43%, and −2.91% [12] in terms of number of external pins, number of external nets, HPWL, and routed wirelength, respectively. Reductions in the number of external pins (and external nets) compared with ISM [12] can be directly attributed to the fact that DOISM seeks to minimize external pin count. However, DOSIM is also able to obtain a reduction in HPWL.

In GPlace3.0, DOISM is applied hierarchically to CLBs, LUT pairs, and FFs. Because GPlace3.0 does not have an initial, explicit packing stage, we do not necessarily start with strong connectivity between LUTs and FFs within the CLBs. However, by using DOISM to minimize HPWL and external pin-count in an alternating fashion, the connectivity between CLBs can be simplified and gradually improved under the influence of both objectives. Moving CLBs not only improves HPWL, but also helps to escape local minima. In addition, minimizing the external pins and nets reduces the workload of the router, as the router has fewer numbers of nets, and nets with fewer pins, to route. (Intuitively, routing an $n$-pin net is typically easier than routing an $m$-pin net, where $m > n$.) Therefore, DOISM achieves better routed wirelength than ISM in GPlace3.0 flow.

In the second experiment, routed wirelength is compared before and after applying DOISM. Table 8 clearly shows that DOISM produces placements with (on average) −12.01% less routed wirelength.

### 5.4 Congestion Estimation

In this section, we compare the performance of mPFGR to other, well-known congestion estimation methods, including WLPA, VPR's PathFinder cost (PFGR), and the negotiation based cost function embedded in the ASIC NCTU Global Router (NCTUgr2.0). Table 9 shows the SAD and AANE values for each of the previous estimation methods for the 12 ISPD benchmarks. The results in

Table 8. Routed Wirelength before andafter Dual-Objective Independent Set Matching (DOISM) Detailed Placement

| Benchmark | Post Global Placement Routed WL[a] | Post DOISM Routed WL | Improvement (%) |
|---|---|---|---|
| FPGA-1 | 463895 | 355720 | −23.32% |
| FPGA-2 | 753476 | 644438 | −14.47% |
| FPGA-3 | 3491431 | 3101107 | −11.18% |
| FPGA-4 | 5949053 | 5402715 | −9.18% |
| FPGA-5 | 11570305 | 10507039 | −9.19% |
| FPGA-6 | 6800940 | 5820321 | −14.42% |
| FPGA-7 | 10604927 | 9508871 | −10.34% |
| FPGA-8 | 8813467 | 8126475 | −7.79% |
| FPGA-9 | 12982007 | 11710738 | −9.79% |
| FPGA-10 | 9382827 | 6836300 | −27.14% |
| FPGA-11 | 11126867 | 10260312 | −7.79% |
| FPGA-12 | 8406145 | 7224371 | −14.06% |
| Total | **90345340** | **79498407** | **−12.01%** |

[a]Routed WL is based on Vivado detailed router.

Table 9. Comparison between Different Congestion Estimation Methods on the 12 ISPD 2016 Benchmark Suite

| | WLPA | | VPR cost | | NCTUgr2.0 cost | | mPFGR cost | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | SAD[a] | AANE | SAD | AANE | SAD | AANE | SAD | AANE |
| FPGA-1 | 438.13 | 0.0101 | 533.04 | 0.0123 | 382.30 | 0.0088 | 385.06 | 0.0089 |
| FPGA-2 | 1054.19 | 0.0234 | 956.02 | 0.0216 | 721.06 | 0.0160 | 721.86 | 0.0160 |
| FPGA-3 | 3315.14 | 0.0506 | 2997.82 | 0.0458 | 1947.83 | 0.0297 | 1948.33 | 0.0297 |
| FPGA-4 | 4816.34 | 0.0664 | 5490.95 | 0.0757 | 2441.59 | 0.0337 | 2440.95 | 0.0336 |
| FPGA-5 | 8861.61 | 0.1003 | 9437.32 | 0.1068 | 3492.89 | 0.0395 | 3407.85 | 0.0385 |
| FPGA-6 | 4455.26 | 0.0680 | 4848.85 | 0.0740 | 3150.70 | 0.0481 | 3146.93 | 0.0480 |
| FPGA-7 | 5656.85 | 0.0666 | 8743.73 | 0.1030 | 3899.12 | 0.0459 | 3887.16 | 0.0458 |
| FPGA-8 | 6804.91 | 0.0924 | 9039.32 | 0.1227 | 4322.54 | 0.0587 | 4329.88 | 0.0588 |
| FPGA-9 | 7697.48 | 0.0915 | 10791.36 | 0.1283 | 4652.12 | 0.0553 | 4622.25 | 0.0549 |
| FPGA-10 | 7397.77 | 0.0857 | 7263.71 | 0.0841 | 6590.56 | 0.0763 | 6541.23 | 0.0758 |
| FPGA-11 | 8405.49 | 0.0972 | 11014.24 | 0.1274 | 4813.60 | 0.0557 | 4811.26 | 0.0557 |
| FPGA-12 | 6796.92 | 0.0872 | 6810.71 | 0.0874 | 5294.54 | 0.0679 | 5255.54 | 0.0674 |

[a]All metrics are relative to the baseline congestion obtained after Vivado detailed router.

Table 9 show that both mFPGR and NCTUgr2.0 always achieve lower SAD and AANE values compared with WLPA and VPR's PathFinder. When compared head-to-head, mFPGR obtains lower SAD and AANE values compared with NCTUgr2.0 for 8 of the 12 benchmarks. In general, mFPGR outperforms NCTUgr2.0 on the larger, more congested benchmarks. The impact of congestion estimation on routability, routed wirelength, and the runtime of the Vivado router will be discussed in Section 5.10.

### 5.5 Window-Based Bi-partition Legalization Effectiveness Validation

Table 10 shows the effectiveness of the WB-legalization approach compared to the non-window-based legalization that uses the entire FPGA. Results obtained in Table 10 clearly indicate that the

Table 10. Routed Wirelength w. and w/o Window-Based Legalization

| Benchmark | w/o Window-Based legalization Routed WL | w. Window-Based Legalization Routed WL | Improvement (%) |
|---|---|---|---|
| FPGA-1 | 388206 | **355720** | **−8.37%** |
| FPGA-2 | 695076 | **644438** | **−7.29%** |
| FPGA-3 | 3939870 | **3101107** | **−21.29%** |
| FPGA-4 | 5627399 | **5402715** | **−3.99%** |
| FPGA-5 | 10674889 | **10507039** | **−1.57%** |
| FPGA-6 | 5934822 | **5820321** | **−1.93%** |
| FPGA-7 | 9561706 | **9508871** | **−0.55%** |
| FPGA-8 | **8062642** | 8126475 | 0.79% |
| FPGA-9 | **11583546** | 11710738 | 1.10% |
| FPGA-10 | 6944954 | **6836300** | **−1.56%** |
| FPGA-11 | 10337550 | **10260312** | **−0.75%** |
| FPGA-12 | 7261613 | **7224371** | **−0.51%** |
| Total | **81012273** | **79498407** | **−1.87%** |

WB-legalization outperforms the non-WB-legalization on average by 1.87% in routed wirelength. The most improvement occurs for the smaller benchmarks (i.e., FPGA01- FPGA04). This is because the area that the inflated cells cover for these smaller benchmarks tends to be much less than the total area available on the FPGA. Working within this smaller area, WB-legalization is able to spread cells out uniformly from the center of each congestion hotspot, thus avoiding potentially large increases in wirelength when seeking to reduce congestion. The same is less true for larger benchmarks. The area occupied by inflated cells tends to be much larger, possibly occupying the entire FPGA. Consequently, cells may have to move much farther from their original positions to avoid congestion, thus adversely affecting wirelength.

### 5.6 Runtime Analysis

The runtime breakdown of GPlace3.0 is shown in Table 11. On average, 68.84% of the total runtime is consumed by flat global placement (34.43% by WL-driven global placement, and 34.41% by congestion-driven global placement), while congestion estimation based on mPFGR and the dual-objective ISM detailed placement (DOISM) consume 13.71% and 17.06% of the total runtime, respectively.

### 5.7 Comparison with Previous Works

In this section, we compare the results achieved by GPlace3.0 with the top three winners of the ISPD'16 placement contest as well as other state-of-the-art FPGA placers. The comparison is based on both routed wirelength and runtime. All routed wirelength are reported by Xilinx Vivado v2015.4. Normalized results in the last row of each table (Tables 12, 13, and 14) are based on the comparisons with GPlace3.0, and only benchmarks that state-of-the-art placers completed successfully are considered. Table 12 compares GPlace3.0 to the three ISPD contest winners using the 12 contest benchmarks in Table 4. Unlike the contest winners, GPlace3.0 finds a routable solution for all 12 benchmarks. Moreover, it obtains the best solution for each benchmark. GPlace3.0 outperforms the top three placers in routed wirelength by 7.53%, 15.15%, and 33.50%, respectively. Even though the top three winners at the recent ISPD'16 FPGA placement contest were run on the same machine and the proposed GPlace3.0 was run on ad different machine, we can still see that the runtime of

Table 11. Runtime Breakdown of GPlace3.0 in (Seconds)

| Benchmark | GP1(s) WL-Driven | GR(s) Global Router | GP2(s) Cong-Driven | DP (s) DOISM | Others (s) | Total (s) |
|---|---|---|---|---|---|---|
| FPGA-1 | 18 | 11 | 18 | 23 | 1 | 71 |
| FPGA-2 | 34 | 21 | 34 | 45 | 2 | 136 |
| FPGA-3 | 163 | 87 | 162 | 111 | 2 | 525 |
| FPGA-4 | 170 | 114 | 171 | 112 | 2 | 569 |
| FPGA-5 | 181 | 170 | 182 | 118 | 3 | 654 |
| FPGA-6 | 369 | 137 | 368 | 182 | 4 | 1060 |
| FPGA-7 | 398 | 174 | 399 | 192 | 4 | 1167 |
| FPGA-8 | 400 | 203 | 399 | 203 | 5 | 1210 |
| FPGA-9 | 551 | 240 | 547 | 240 | 5 | 1583 |
| FPGA-10 | 588 | 134 | 595 | 264 | 6 | 1587 |
| FPGA-11 | 523 | 194 | 518 | 243 | 5 | 1483 |
| FPGA-12 | 732 | 159 | 732 | 312 | 7 | 1942 |
| Norm. | **34.43%** | **13.71%** | **34.41%** | **17.06%** | **0.38%** | **100%** |

Table 12. Comparison with ISPD'16 Contest Winners on ISPD 2016 Benchmark Suite

| | 1st Place (UTPlaceF1.0) | | 2nd Place (RippleF1.0) | | 3rd Place (GPlace1.0) | | GPlace3.0 | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | Routed WL | Runtime(s) | Routed WL | Runtime(s) | Routed WL | Runtime(s) | Routed WL | Runtime(s) |
| FPGA-1 | PE[a] | - | 379932 | 118 | 581975 | 97 | **355720** | **71** |
| FPGA-2 | 677877 | 435 | 679878 | 208 | 1046859 | 191 | **644438** | **136** |
| FPGA-3 | 3223042 | 1527 | 3660659 | 1159 | 5029157 | 862 | **3101107** | **525** |
| FPGA-4 | 5628519 | 1257 | 6497023 | 1149 | 7247233 | 889 | **5402715** | **569** |
| FPGA-5 | 10264769 | 1266 | UR | - | UR | - | **10507039** | **654** |
| FPGA-6 | 6630179 | 2920 | 7008525 | 4166 | 6822707 | 8613 | **5820321** | **1060** |
| FPGA-7 | 10236827 | 2703 | 10415871 | 4572 | 10973376 | 9169 | **9508871** | **1167** |
| FPGA-8 | 8384338 | 2645 | 8986361 | 2942 | 12299898 | 2741 | **8126475** | **1210** |
| FPGA-9 | UR[b] | - | 13908997 | 5833 | UR | - | **11710738** | **1583** |
| FPGA-10 | PE | - | PE | - | UR | - | **6836300** | **1587** |
| FPGA-11 | 11091383 | 3227 | 11713479 | 7331 | UR | - | **10260312** | **1483** |
| FPGA-12 | 9021769 | 4539 | PE | - | UR | - | **7224371** | **1942** |
| Norm. | **+7.53%** | **2.35X** | **+15.15%** | **3.52X** | **+33.50%** | **4.76X** | **0.00%** | **1.00X** |

[a]PE: Placement error.
[b]UR: Unroutable placement.

GPlace3.0 is faster by 2.35X, 3.52X, and 4.76X than these placers, respectively. Table 13 compares GPlace3.0 to GPlace2.0 [17]. Unlike GPlace2.0 [17], GPlace3.0 finds a routable solution for all 12 benchmarks. Moreover, it obtains the best solution for each benchmarks. GPlace3.0 outperforms GPlace2.0 [17] in routed wirelength by 20.47%, with a slight increase in CPU time.

Table 14 compares GPlace3.0 to the most recently improved versions of the first- and second-place contest winners RippleF2.0 [18], UTPlaceF2.0 [13], and UTPlaceF3.0 [12]. It is clear that GPlace3.0 outperforms all placers in routed wirelength on the majority of the benchmarks (8 out of 12 benchmarks) and achieves the best overall routed wirelength and runtime. All runtime results reported in this article for RippleF2.0 [18], UTPlaceF2.0 [13], and UTPlaceF3.0 [12] are based on binaries received from the authors of [12], [13], and [18]. GPlace3.0 outperforms RippleF2.0 [18],

Table 13. Comparison with GPLace2.0 [17] on ISPD 2016 Benchmark Suite

| | GPlace2.0 [17] | | GPlace3.0 | |
|---|---|---|---|---|
| Benchmark | Routed WL | Runtime(s) | Routed WL | Runtime(s) |
| FPGA-1 | 493788 | **30** | **355720** | 71 |
| FPGA-2 | 903099 | **61** | **644438** | 136 |
| FPGA-3 | 3908244 | **289** | **3101107** | 525 |
| FPGA-4 | 6277878 | **280** | **5402715** | 569 |
| FPGA-5 | UR[a] | - | **10507039** | 654 |
| FPGA-6 | 7643382 | **600** | **5820321** | 1060 |
| FPGA-7 | 11255351 | **691** | **9508871** | 1167 |
| FPGA-8 | 9323360 | **734** | **8126475** | 1210 |
| FPGA-9 | 14002965 | **974** | **11710738** | 1583 |
| FPGA-10 | UR | - | **6836300** | 1587 |
| FPGA-11 | 12367773 | **923** | **10260312** | 1483 |
| FPGA-12 | UR | - | **7224371** | 1942 |
| Norm. | **+20.47%** | **0.59X** | **0.00%** | **1.00X** |

[a]UR: Unroutable placement.

Table 14. Comparison with State-of-the-Art Academic FPGA Placers on ISPD 2016 Benchmark Suite

| | RippleF2.0 [18] | | UTPlaceF2.0 [13] | | UTPlaceF3.0 [12] | | GPlace3.0 | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | Routed WL | Runtime(s) | Routed WL | Runtime(s) | Routed WL | Runtime(s) | Routed WL | Runtime(s) |
| FPGA-1 | 362563 | 74 | 384709 | 195 | 356769 | 152 | **355720** | **71** |
| FPGA-2 | 677563 | 167 | 652690 | 350 | **642108** | 244 | 644438 | **136** |
| FPGA-3 | 3617466 | 1037 | 3181331 | 1271 | 3215087 | 659 | **3101107** | **525** |
| FPGA-4 | 6037293 | 621 | 5504083 | 1117 | 5409765 | 635 | **5402715** | **569** |
| FPGA-5 | 10455204 | 1012 | 10068879 | 1189 | **9659958** | 803 | 10507039 | **654** |
| FPGA-6 | 6960037 | 2772 | 6411247 | 2524 | 6487628 | 1360 | **5820321** | **1060** |
| FPGA-7 | 10248020 | 2170 | 10040562 | 2429 | 10104837 | 1373 | **9508871** | **1167** |
| FPGA-8 | 8874454 | 1426 | 8113483 | 2232 | **7879022** | 1321 | 8126475 | **1210** |
| FPGA-9 | 12954350 | 2683 | 13616625 | 2925 | 12369055 | 2061 | **11710738** | **1583** |
| FPGA-10 | 8564363 | 5555 | 8866049 | 3503 | 8794515 | 2626 | **6836300** | **1587** |
| FPGA-11 | 11226088 | 3636 | 10834629 | 2900 | **10196038** | 1671 | 10260312 | **1483** |
| FPGA-12 | 8928528 | 9748 | 8246410 | 4338 | 7755443 | 2335 | **7224371** | **1942** |
| Norm. | **+11.83%** | **2.58X** | **+8.08%** | **2.08X** | **+4.24%** | **1.15X** | **0.00%** | **1.00X** |

UTPlaceF2.0 [13], and UTPlaceF3.0 [12] in routed wirelength by 11.83%, 8.08%, and 4.24%, respectively. In terms of runtime, GPlace3.0 is about 2.58X, 2.08X, and 1.15X faster than RippleF2.0 [18], UTPlaceF2.0 [13], and UTPlaceF3.0 [12], respectively.

Looking more closely at the results in Table 14, we can observe that for FPGA-10 benchmark, GPlace3.0 outperforms UTPlaceF3.0 [12] by 22.27% in routed wirelength, while for FPGA-05 benchmark, UTPlaceF3.0 [12] outperforms GPlace3.0 by 8.06% in routed wirelength.

To understand this difference in performance, we first remind the reader of the relevant features of the two benchmarks. FPGA-05 is a small-to-medium size benchmark with 174K FFs, 264K LUTs, and 1281 control sets. It is also a highly-connected circuit, as witnessed by its high Rent exponent (i.e., 0.8). FPGA-10, on the other hand, is a large benchmark with 600K FFs, 264K LUTs, and 2541 control sets. However, its Rent exponent is lower (i.e., 0.6), indicating that it is less connected compared to FPGA-05.

Table 15. Vivado Router CPU Time in (Seconds): GPlace3.0 vs. UTPlaceF3.0 [12] on 12 ISPD16 Benchmarks

| Benchmark | GPlace3.0 Router Runtime(s) | UTPlaceF3.0 [12] Router Runtime(s) | Improvement (%) |
|---|---|---|---|
| FPGA-1 | 126 | **121** | −3.97% |
| FPGA-2 | 137 | **135** | −1.46% |
| FPGA-3 | 286 | **255** | −10.84% |
| FPGA-4 | 320 | **290** | −9.38% |
| FPGA-5 | **2527** | 3921 | 55.16% |
| FPGA-6 | **425** | 555 | 30.59% |
| FPGA-7 | **655** | 2269 | 246.41% |
| FPGA-8 | 442 | **375** | −15.16% |
| FPGA-9 | **739** | 1094 | 48.04% |
| FPGA-10 | **622** | 1274 | 104.82% |
| FPGA-11 | **763** | 2036 | 166.84% |
| FPGA-12 | **670** | 882 | 31.64% |
| Total | **7712** | **13207** | **71.25%** |

In general, the high number of control sets in FPGA-10 significantly affects how FFs can be packed into slices without violating legality rules. This is problematic for UTPlaceF3.0 [12], because the FIP used to guide packing does not handle control-set constraints. This causes the packer to group FFs that may be physically far apart just to achieve a feasible packing. The result is a packed netlist that may be "wirelength unfriendly" from the start. In contrast, GPlace3.0 avoids this problem by not performing (early) packing, and optimizing globally (in Phases I and II) while enforcing all legalization constraints. Moreover, in Phase III of the GPlace3.0 flow, the lesser connectivity in FPGA-10 allows DOISM to more easily improve the final placement of FFs and CLBs with an eye towards optimizing wirelength and routability.

In the case of FPGA-05, the fewer number of FFs and control sets makes the circuit much easier to pack into slices without violating legality rules. This fact, coupled with the fact that FPGA-05 is more connected than FPGA-10, benefits the packing approach in UTPlaceF3.0 [12]. This is because many external nets can be easily absorbed early on to produce a netlist that is less congested. In contrast, GPlace3.0, which uses cell inflation to alleviate congestion, may inflate cells more than necessary due to the abundance of interconnections, in which case DOISM may not be able to reduce the excess wirelength during detailed placement.

### 5.8 Vivado Router Runtime Comparisons

Table 15 compares Xilinx Vivado router runtime on placements produced by GPlace3.0 to those produced by UTPlaceF3.0 [12]. GPlace3.0 produces placements that require on average less effort by Vivado router and can be routed on average faster by 71.25% than those produced by UTPlaceF3.0 [12]. This indicates that the quality of placements produced by GPlace3.0 are less congested and of better quality than those produced by UTPlaceF3.0 [12]. Therefore, the compilation time of GPlace3.0 flow on average is much better than that of UTPlaceF3.0 [12].

### 5.9 Timing Results Comparisons

Although the current implementation of GPlace3.0 is non-timing driven, we include timing results in Table 16 and compare them with the other state-of-the-art routability-driven placers (which are

Table 16. Timing Results Compared with State-of-the-Art Academic FPGA Placers on ISPD 2016 Benchmark Suite

| | RippleF2.0 [18] | | | | UTPlaceF3.0 [12] | | | | GPlace3.0 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | MHz | R-WL | FMax Ratio | R-WL Ratio | MHz | R-WL | FMax Ratio | R-WL Ratio | MHz | R-WL | FMax Ratio | R-WL Ratio |
| FPGA-1 | 125.0 | 435583 | 0.89 | 1.08 | **153.8** | 437373 | **1.09** | 1.09 | 140.8 | **402469** | 1.00 | **1.00** |
| FPGA-2 | 107.5 | 752299 | 0.80 | 1.04 | 133.3 | 757241 | 0.99 | 1.05 | **135.1** | **720714** | **1.00** | **1.00** |
| FPGA-3 | 78.1 | 4033650 | 0.95 | 1.16 | **89.3** | 3893436 | **1.08** | 1.12 | 82.6 | **3473792** | 1.00 | **1.00** |
| FPGA-4 | 59.5 | 6608994 | 0.86 | 1.12 | **70.9** | 6043884 | **1.02** | 1.03 | 69.4 | **5881605** | 1.00 | **1.00** |
| FPGA-5 | 38.3 | 10857194 | 0.89 | 0.99 | 40.3 | **10319925** | 0.93 | **0.94** | **43.3** | 10963581 | **1.00** | 1.00 |
| FPGA-6 | 50.0 | 7542706 | 0.90 | 1.15 | 52.4 | 7806056 | 0.94 | 1.19 | **55.6** | **6558411** | **1.00** | **1.00** |
| FPGA-7 | 38.6 | 11489394 | 0.84 | 1.13 | 27.9 | 10695627 | 0.80 | 1.05 | **46.1** | **10187951** | **1.00** | **1.00** |
| FPGA-8 | 69.0 | 9662019 | 0.96 | 1.12 | **73.0** | 8818583 | **1.01** | 1.02 | 71.9 | **8642408** | 1.00 | **1.00** |
| FPGA-9 | 31.4 | 14646682 | 0.69 | 1.17 | 42.4 | 13470206 | 0.92 | 1.08 | **45.9** | **12486925** | **1.00** | **1.00** |
| FPGA-10 | 48.1 | 9122777 | 0.81 | 1.20 | 17.2 | 9650067 | 0.29 | 1.27 | **59.2** | **7614759** | **1.00** | **1.00** |
| FPGA-12 | 42.9 | 9522550 | 0.87 | 1.22 | **53.8** | 9085992 | **1.09** | 1.16 | 49.5 | **7820768** | 1.00 | **1.00** |
| **geomean** | 57.0 | 5341755.8 | 0.85 | 1.12 | 57.3 | 5166029.2 | 0.86 | 1.09 | **66.7** | **4751988.1** | **1.00** | **1.00** |
| FPGA-11 | NA[a] | - | - | - | NA | - | - | - | **40.8** | **10706748** | **1.00** | **1.00** |

[a]NA: Clock Time period is greater than 200ns.

also non-timing driven). Table 16 shows that GPlace3.0 produces better timing results for most of the benchmarks compared to the other placers. In Table 16, the R-WL columns report post-routed wirelength, the MHz columns give the circuit speed in MHz, and the Fmax Ratio and R-WL Ratio columns provide the ratio of the state-of-the-art placers UTPlace3.0 [12] and Ripple2.0 [18] to GPlace3.0 for maximum frequency (Fmax) and routed wirelength, respectively. The last row of Table 16 gives the geometric mean accross all benchmarks. From Table 16, we see that UTPlace3.0 [12] produces routable placements with 9% higher routed wirelength and 14% worse Fmax compared to GPlace3.0. The placer Ripple2.0 [18] produces routable placements with 12% higher routed wirelength and 15% worse Fmax compared to GPlace3.0. Table 16 shows that on benchmark FPGA11 (i.e., a high resource utilization benchmark with high Rent exponent) both UTPlace3.0 [12] and Ripple2.0 [18] produce unroutable placements, even when the circuit runs at a very low clock speed (e.g., less than 5MHz). GPlace3.0, on the other hand, produces a routable placement with Fmax equal to 40.8MHz for this benchmark.

### 5.10 Extended Experimental Results

To further demonstrate the effectiveness of GPlace3.0, experiments are conducted using an additional 360 benchmarks generated by Xilinx Inc. using Gnl [9]. For each of the 12 benchmarks listed in Table 4, 30 new circuits are generated by varying the original circuit's parameters (i.e., the values shown in columns 2–7 of Table 4). This results in a total of 372 circuits (i.e., 12+12x30) being available. (The ranges of the circuit parameters were shown earlier in Table 5.)

*5.10.1 The Impact of Different GPlace3.0 Modules on Routability.* In this section, we use several metrics to provide a breakdown of the impact of each individual GPlace3.0 module. Table 17 compares the solution quality obtained by using mPFGR versus WLPA to estimate congestion. Column 2 shows that when WLPA is used to estimate congestion, the Vivado router is unable to route 19 of the 372 benchmarks. However, only 6 benchmarks fail to route when using mFPGR. Column 3 shows that WLPA results in placements with 2.08% more post-routing wirelength compared to mFPGR. Finally, column 4 reveals that the Vivado router is able to route placements produced using

Table 17. Performance Comparison with mPFGR versus WLPA

| Congestion Method | #Failures | Routed-WL (Norm.) | Router Runtime (Norm.) |
|---|---|---|---|
| mPFGR | 6 | +0.00% | 1.00X |
| WLPA | 19 | +2.08% | 1.14X |

Table 18. Performance Comparison with and without WB-Legalization

| | #Failures | Routed-WL (Norm.) | Router Runtime (Norm.) |
|---|---|---|---|
| w. WB-Legalization | 6 | +0.00% | 1.000X |
| w/o WB-Legalization | 17 | +2.77% | 0.997X |

Table 19. Performance Comparison of DOISM versus Post Global Placement

| | #Failures | Routed-WL (Norm.) | Router Runtime (Norm.) |
|---|---|---|---|
| Post Global Placement | 13 | +13.47% | 1.08X |
| Post DOISM | 6 | +0.00% | 1.00X |

Table 20. Comparison with the State-of-the-Art on 372 Xilinx Benchmarks

| Placer | #Failures | Success Rate (%) | #Best | Routed-WL (Norm.) | Place Runtime (Norm.) | Route Runtime (Norm.) |
|---|---|---|---|---|---|---|
| RippleF2.0 [18] | 36 | 90.32% | 3 | +13.97% | 3.20X | 1.56X |
| UTPlaceF3.0 [12] | 55 | 85.22% | 103 | +5.19% | 1.15X | 1.77X |
| GPlace3.0 | 6 | **98.39%** | **260** | 0.00% | 1.00X | 1.00X |

mPFGR 1.14x faster compared with placements produced with WLPA. Table 18 compares the solution quality obtained when using the proposed WB-legalization versus non-WB-legalization (that uses the entire area of the FPGA). Results obtained in Table 18 clearly show that when non-WB-legalization is used, the Vivado router is unable to route 17 benchmarks. However, only six benchmarks fail to route when WB-legalization is used. Moreover, non-WB-legalization results in placements with 2.77% more post-routing wirelength compared to WB-legalization. The Vivado router (average) runtime is very similar when using either WB-legalization or non-WB-legalization. Table 19 compares the solution quality obtained using DOISM versus post-global placement. The results in Table 19 show that DOISM reduces the number of unroutable placement solutions from 13 to 6 compared with post-global placement. Moreover, using DOISM produces placements that have 13.47% less post-routing wirelength, and the Vivado router is able to route these placements 1.08x faster compared to the placements obtained using post-global placement.

*5.10.2 Comparisons with State-of-the-Art Placers.* GPlace3.0 is compared with the state-of-the-art placers in References [12] and [18] based on the following metrics: (1) number of times each placer produces a placement that cannot be routed by Vivado router, (2) the percentage of successfully placed and routed circuits, (3) the number of times each placer produces a placement with the best routed wirelength, (4) the normalized routed wirelength of other placers compared with GPlace3.0, (5) the normalized placement runtime of other placers compared to GPlace3.0, and (6) the normalized runtime for the (Vivado) router compared to GPlace3.0. Table 20 shows that

GPlace3.0 fails on only 6 benchmarks, whereas RippleF2.0 [18] and UTPlaceF3.0 [12] fail on 36 and 55 benchmarks, respectively. Note that all three placers fail on the same six benchmarks, which have very high Rent exponent. In terms of QoR, GPlace3.0 produces the best routed wirelength for 260 benchmarks, whereas UTPlaceF3.0 [12] and RippleF2.0 [18] produce the best results for 103 and 3 benchmarks, respectively. GPlace3.0 outperforms UTPlaceF3.0 [12] and RippleF2.0 [18] by 5.19% and 13.97% in routed wirelength, respectively. In terms of placement runtime, GPlace3.0 is 1.15X and 3.20X faster than UTPlaceF3.0 [12] and RippleF2.0 [18], respectively. The last column in Table 20, shows that Xilinx Vivado router is able to route placements produced by GPlace3.0 1.77× and 1.56× faster than placements produced by UTPlaceF3.0 [12] and RippleF2.0 [18], respectively. Overall, GPlace3.0 is able to produce placement solutions that are less congested and of better quality than those produced by UTPlaceF3.0 [12] and RippleF2.0 [18], while requiring much less total placement-and-routing time to do so.

## 6 CONCLUSIONS

With the ever-increasing size and complexity of FPGAs and the designs targeted for them, fast, routability-driven placement is becoming crucial to the modern FPGA CAD flow. In this article, a novel routability-driven analytic placer for modern Xilinx UltraScale FPGAs is presented. The proposed placer seeks to optimize both wirelength and routability. A novel window-based procedure for satisfying legality constraints in lieu of packing, a global router for accurately estimating congestion, and a detailed placement that optimizes both wirelength and routability are all proposed. The performance of the placer is compared to that of (i) the three ISPD 2016 placement contest winners using the original 12 placement benchmarks and (ii) to the most recent (improved) versions of the contest winners using an additional 360 benchmarks provided by Xilinx Inc. Experimental results show that the performance of the proposed placer is superior in terms of routed wirelength, placement runtime, and routing runtime, as well as in terms of the number of routable placements, the fewest non-routable placements, and the number of best placements. Our future work will focus on the areas of congestion estimation and timing. First, we will seek to develop a fast and accurate congestion estimation method based on machine learning, and use the subsequent predictions produced by this method to perform dynamic cell inflation multiple times. Second, we will implement a fully timing-driven version of GPlace3.0.

## REFERENCES

[1] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*. Springer, 213–222.

[2] Huimin Bian, Andrew C. Ling, Alexander Choong, and Jianwen Zhu. 2010. Towards scalable placement for FPGAs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 147–156.

[3] Elaheh Bozorgzadeh, S. Ogrenci Memik, Xiaojian Yang, and Majid Sarrafzadeh. 2004. Routability-driven packing: Metrics and algorithms for cluster-based FPGAs. *Journal of Circuits, Systems, and Computers* 13, 1 (2004), 77–100.

[4] S. Chen and Y. Chang. 2015. Routing-architecture-aware analytical placement for heterogeneous FPGAs. In *Proceedings of the Design Automation Conference*. ACM, 27.

[5] Yu-Chen Chen, Sheng-Yen Chen, and Yao-Wen Chang. n.d. Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*. 647–654.

[6] GNL. n.d. Netlist-Generator Tool. Retrieved 1999 from http://users.elis.ugent.be/dstrooba/gnl/.

[7] Padmini Gopalakrishnan, Xin Li, and Lawrence Pileggi. 2006. Architecture-aware FPGA placement using metric embedding. In *Proceedings of the 43rd Annual Design Automation Conference*. ACM, 460–465.

[8] Marcel Gort and Jason Helge Anderson. 2012. Analytical placement for heterogeneous FPGAs. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. IEEE, 143–150.

[9] G. Grewal, S. Areibi, M. Westrik, Z. Abuowaimer, and B. Zhao. 2017. Automatic flow selection and quality-of-result estimation for FPGA placement. In *Proceedings of the 24th Reconfigurable Architectures Workshop*. Orlando, Florida, 115–123.

[10] W. How, H. Yu, X. Hong, Y. Cai, W. Wu, J. Gu, and W. Kao. 2001. A new congestion-driven placement algorithm based on cell inflation. In *Proceedings of the Design Automation Conference, Asia and South Pacific*. IEEE, 605–608.
[11] J. Hu, J. A Roy, and I. Markov. 2010. Completing high-quality global routes. In *ISPD*. ACM, 35–41.
[12] W. Li, S. Dhar, and D. Pan. 2017. UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).
[13] Wuxi Li, Shounak Dhar, and David Z. Pan. 2016. UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'16)*. 66:1–66:7.
[14] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. 2013. NCTU-GR 2.0: Multithreaded collision-aware global routing with bounded-length maze routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 5 (2013), 709–722.
[15] Larry McMurchie and Carl Ebeling. n.d. PathFinder: A negotiation-based performance-driven router for FPGAs. In *Proceedings of the 1995 ACM 3rd International Symposium on Field-Programmable Gate Arrays*. 111–117.
[16] M. Pan, Y. Xu, Y. Zhang, and C. Chu. 2012. FastRoute: An efficient and high-quality global router. *VLSI Design* (2012), 1–19.
[17] R. Pattison, Z. Abuowaimer, S. Areibi, G. Grewal, and A. Vannelli. 2016. Invited paper: GPlace—A congestion-aware placement tool for UltraScale FPGAs. In *Proceedings of the International Conference on Computer Aided Design*. Austin, Texas, 1–7.
[18] C. Pui, G. Chen, W. Chow, K. Lam, P. Tu, H. Zhang, E. Young, and B. Yu. 2016. RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. 1–8.
[19] Peter Spindler and Frank M. Johannes. 2007. Kraftwerk: A fast and robust quadratic placer using an exact linear net model. In *Modern Circuit Placement*. Springer, 59–93.
[20] Russell Tessier and Heather Giza. 2000. Balancing logic utilization and area efficiency in FPGAs. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*. Springer, 535–544.
[21] Marvin Tom, David Leong, and Guy Lemieux. 2006. Un/DoPack: Re-clustering of large system-on-chip designs with interconnect variation for low-cost FPGAs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 680–687.
[22] D. Xie, J. Xu, and J. Lai. 2009. A new FPGA placement algorithm for heterogeneous resources. In *Proceedings of the ASICON'09 Conference*. 742–746.
[23] Xilinx. n.d. ISPD 2016 Routability-Driven FPGA Placement Contest. Retrieved March 17, 2017 from http://www.ispd.cc/contests/16/ispd2016_contest.html.
[24] Xilinx. [n. d.]. UltraScale Architecture Configurable Logic Block User Guide. http://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.
[25] M. Xu, G. Grewal, and S. Areibi. 2011. StarPlace: A new analytic method for FPGA placement. *Integration, The VLSI Journal* 44, 3 (June 2011), 192–204.
[26] Yonghong Xu and Mohammed A. S. Khalid. 2005. QPF: Efficient quadratic placement for FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, 555–558.
[27] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal. 2016. Routability-driven FPGA placement contest. In *Proceedings of the International Symposium on Physical Design*. ACM, 139–143.
[28] D. Yeager, D. Chiu, and G. Lemieux. 2007. Congestion estimation and localization in FPGAs: A visual tool for interconnect prediction. In *Proceedings of the International Workshop on System Level Interconnect Prediction*. ACM, 33–40.

Received November 2017; revised April 2018; accepted June 2018